

琢石成器

Windows 环境下 32位汇编语言程序设计

罗云彬 著



琢石成器——Windows 环境下 32 位 汇编语言程序设计

书名:	责编:
社名:	校次:
开本:	正文页码:
文前页码:	排版员:
日期:	排版公司:
主管签字:	

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

Windows 环境下 32 位汇编语言是一种全新的编程语言。它使用与 C++ 语言相同的 API 接口，不仅可以开发出大型的软件，而且是了解操作系统运行细节的最佳方式。

本书从编写应用程序的角度，从“Hello, World!”这个简单的例子开始到编写多线程、注册表和网络通信等复杂的程序，通过 70 多个实例逐步深入 Win32 汇编语言编程的方方面面。

本书作者罗云彬拥有十余年汇编语言编程经验，是汇编编程网站 <http://www.win32asm.com.cn> 的创办者。本书是作者多年来编程工作的总结，适合于欲通过 Win32 汇编语言编写 Windows 程序的读者。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

琢石成器：Windows 环境下 32 位汇编语言程序设计 / 罗云彬著. —北京：电子工业出版社，2009.6
ISBN 978-7-121-08663-2

I. 琢… II. 罗… III. 汇编语言—程序设计 IV. TP313

中国版本图书馆 CIP 数据核字（2009）第 059467 号

责任编辑：李 冰

印 刷：北京东光印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：47.25 字数：1030 千字

印 次：2009 年 6 月第 1 次印刷

印 数：4000 册 定价：89.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言

从 Windows 出现开始，汇编语言似乎在慢慢地销声匿迹，但本书可以让人放弃这个观点，其实在 Win32 环境下，汇编语言依然强大。

Why——为什么选择 Win32 汇编

选择 Win32 汇编的理由是什么呢？

在 DOS 时代，学习汇编就是学习系统底层编程的代名词，仅要成为一名入门级的汇编程序员，就需要学习从 CPU 结构、CPU 工作方式、各种硬件的编程方法到 DOS 工作方式等范围很广的知识。随着 Windows 时代的到来，Windows 像一堵巨大的墙，把我们和计算机的硬件隔离开。对于 DOS 的汇编程序员来说，就像在一夜之间，我们发现自己曾经学过的几乎所有的东西都被 Windows 封装到内核中去了，由于保护模式的存在，我们又无法像在 DOS 下那样闯入系统内核为所欲为。在 Windows 下用任何语言编程都必须遵循 Windows 的规范，汇编也不例外，也就是说，汇编不再是一种“有特权”的语言。面对汹涌而来的 Visual C++，Visual Basic，PowerBuilder 和 Java 等各个领域的猛将，从 DOS 时代“为所欲为”的“系统警察”岗位下岗，在其他领域又没有一技之长，汇编语言似乎失去了生存的意义，有很多人在 DOS 转向 Windows 的时候放弃了汇编语言。

但是经过短暂的失落，摆正了自己在系统中的位置，我们发现从“系统警察”转换到遵循 Windows 规范的“好市民”后，汇编语言又慢慢地在这个世界流行起来了。毕竟，不能为所欲为也可以有好的一面，我们可以不必再考虑一些老大难的问题，如程序运行时面对什么样的显示卡，如何驱动不同的打印机，内存不够了如何用磁盘交换，等等。我们也可以在了解更少硬件知识的情况下就可以掌握 Win32 的汇编编程。而且，我们惊喜地发现，做了“好市民”以后，我们反而拥有了和其他语言同样的权利——为了做图形和界面等方面的功能，汇编程序员在 DOS 时代连做梦都在羡慕 C 语言庞大的函数库，而现在，Windows 为我们提供了比这还要多得多的函数，以至于其他大部分语言可以做出来的功能，汇编都可以做，而其他语言做不到的功能，汇编照样可以做！所以这就是理由之一：**Win32 汇编可以当做一种功能强大的开发语言使用，使用它完全可以开发出大型的软件来。**

正因为 Win32 汇编看上去不再那样低级，于是有读者曾经提出：Win32 汇编讲的都是用 API 来写程序，和高级语言差不多，以前在 DOS 下使用的中断什么的都不能用，所以没有什么新奇的了。还有读者认为本书只不过是 MSDN 的汇编版本而已。言下之意就是：学汇编就是为了了解高级语言底下一层的功能，但现在 Win32 汇编却使用 and C++ 等语言相同的 API 接口，既然和高级语言处于同一个级别，我们为什么还要去和机器指令打交道呢，还不如去学 Visual C++ 方便。

但是我们可以这样问一问自己：

问：在 DOS 汇编中我们为什么用中断功能？

答：为了使用 DOS 内核提供的功能。

问：在 DOS 中我们常常自己用操作 I/O 端口的方法读写硬盘或操作显卡吗？

答：不，我们用系统提供的 int 13h 和 int 10h。

.....

同样，在 Win32 汇编里使用 API 也是为了使用 Windows 内核提供的功能。只不过使用的方式不再是中断方式而已，这不是 Win32 汇编语言“高级化”了，而是高级语言因为使用 Windows 的 API 接口而“低级化”了，其代价就是无法移植到其他系统，用 Visual C++ 写的程序是无法移植到其他操作系统平台上的，只有和平台无关的 ANSI C++ 等才能算是真正意义上的高级语言。

其实，任何汇编语言都是和操作系统密切相关的，不管是 DOS 汇编、Win32 汇编，还是 Linux 汇编，都是基于特定的操作系统的，如果一定要绕过操作系统，那么就不会有 DOS 汇编和 Win32 汇编的区别了，但是这样的话我们不是在学汇编，而是在自己开发操作系统。高级语言在不同的操作系统上看起来都差不多，但作为一种低级语言，不同操作系统上的汇编就是不同的世界。所以，既然 Windows 和 DOS 是两个完全不同的操作系统，我们就必须抛弃 DOS 汇编中的大部分概念从头开始学习 Win32 汇编。这就是理由之二：**Win32 汇编是 Windows 环境下一种全新的编程语言。**

Win32 环境下的很多高级语言，如 Visual C++ 和 Visual Basic 等，一如既往地实现的细节进行了或深或浅的封装，就连最能表现 Windows 特征的部分，如消息循环和多线程的处理等内容也都被隐藏封装，使我们在使用它们进行可视化编程的同时，无法全面了解 Win32 程序运行的具体方式。在学习 Win32 汇编以后，这些隐藏在高级语言后面的细节就暴露出来了。

由于封装的关系，各种高级语言或多或少存在某种“缺陷”，比如 VB 不支持指针，结果很多需要使用指针的 API 用起来就很不方便，像多线程一类的特征在 VB 中就无法实现，PowerBuilder 也是如此；C 语言已经是最灵活的高级语言了，但还是无法在代码级别处理某些需求；而汇编语言见到的是一个最真实的操作系统，它可以用最灵活的方式使用各种系统功能，第 13 章中有关进程隐藏的内容就是最好的写照。所以理由之三就是：**使用 Win32 汇编语言是了解操作系统运行细节的最佳方式。**

最后的理由根本不是理由，而是必然的选择，当我们在 Windows 环境下进行加密解密、逆向工程，还有病毒、木马等有害代码的分析和防治工作时，Win32 汇编是唯一的选择。在任何讨论这方面内容的书籍中，汇编代码的篇幅总是很大的。因此，要想深入了解这些内容的前提就是深入汇编编程。

How——如何学习 Win32 汇编

以往的汇编书籍往往把重点放在硬件结构和指令上，讲述了一大堆电路框图和指令列

表，把大家搞得晕晕乎乎后，再举出一些重量级的例子，不是一些像数组、矩阵计算一类的复杂运算，就是开始图形模式画图，以至于大家看完以后就再也找不到北了！实际上，这些例子不是太难了，而是太枯燥了。有人说，学汇编就像考大学，千军万马过独木桥，太多的人中途放弃了，只有少数人坚持到最后。

笔者认为：学习汇编应该在轻松的环境下进行，在学习中使用的例子不一定太复杂，但一定要有吸引力。用汇编写复杂的运算程序固然会比 C 更有效率，但同样的事在 C 中用一个表达式就全部搞定了，从这里开始学汇编，给人的感觉就像从复杂的公式开始学算术，要知道，加法还没有学会呢！而对于高级语言封装起来的系统功能，用汇编解释起来就非常直接，非常自然，也更容易懂。以笔者自己学汇编的过程来说，那时候是 1990 年，刚好是中国第一次病毒大流行，大家的计算机上都是那个病毒的开山鼻祖——乒乓病毒，在流行 DOS 的时期，看着在屏幕上蹦的小球，心中就有一个问题：如何编出这样一个玩意来呢？要知道 DOS 是单任务的，而那个球在别的程序运行的时候照样蹦！这用当时流行的 FORTRAN、C 等课程中学到的任何知识都无法解释，因为这些课程中不可能有 TSR、中断、引导区等内容。带着这样一个疑问学习汇编，在分析乒乓病毒的过程中啃一条条不懂的指令，病毒分析完了，汇编课也学完了，而且反过来看那些复杂的计算程序都是那么顺理成章，不攻自破了。实际上，从一些实用的系统功能开始学习汇编远比学矩阵计算容易理解。

正如最经典的 C 程序就是那个“Hello, World!”一样，这个程序的有名并不是因为它用高深复杂的语句放倒了一大批人，而是它以最简单易懂的方式让人们走入 C 语言的大门。对于 Win32 汇编也是如此，从最简单的例子开始总是没错的，笔者建议读者跟随本书中从简到繁的例子，努力做到理解并灵活引用这些例子中的各种功能，正如“熟读唐诗三百首，不会写诗也会吟”，最后能够熟练地使用 Win32 汇编来解决各种编程需求就是最大的胜利。

另外，正如前面讲到的，汇编语言的学习必须和操作系统紧密结合。经过简单的调查，笔者发现很多高校使用的汇编教程还是停留在清华 91 版《IBM-PC 汇编语言程序设计》之类的教材上，虽然这些教材中基础知识部分永远不会过时，但涉及操作系统的部分还是停留在 DOS 阶段。随着 DOS 操作系统的悄然引退，继续把精力花在上面是一种浪费，因为任何语言都必须有应用的平台，否则课程学完之后会尴尬地发现没有地方可以应用。笔者认为，在《IBM-PC 汇编语言程序设计》之类传统教材中的基础部分学习完毕以后，重点就应该转向 Win32 汇编，以及保护模式方面的知识。

关于本书的内容

本书尝试从编写应用程序的角度，从“Hello, World”这个简单的例子开始到编写多线程、注册表和网络通信等复杂的程序，通过 70 多个从简单到复杂的例子，逐步深入 Win32 汇编编程的方方面面。笔者从事汇编编程已经有十几年的历史了，从 8086 时代的 DOS 汇编编程开始到当前的 Win32 汇编编程，从一个初学者到现在能利用 Win32 汇编来解决大部

分编程需求，中间也经过了很长时间的摸索和大量的挫折，所以笔者很清楚初学者在哪些地方会遇到问题，但是涉及 Win32 汇编的书籍却实在太少了。正是因为如此，笔者决心把本书的目标定为：能让读者入门并在最后能熟练掌握 Win32 汇编编程，而不是那种深入系统奥秘一类的书籍。

从这个目标出发，本书的选材中尽量去掉已经有其他书籍详细讨论的部分，因为要一本书涉及全部方面是不现实的。内容全面就必然不精，内容深刻就必须围绕一个中心点，所以本书的内容并不详细讨论一般汇编教材的基础部分，如处理器结构和保护模式等，也不准备涉及 Windows 驱动程序、COM 编程或者其他能够冠以“密技”头衔的内容。本书主要的内容将放在 32 位宏汇编对比 DOS 汇编所不同的部分，以及 Win32 应用程序的汇编实现上。不求全面，只求精也！（说句老实话，也不敢对自己不精通的地方妄加评论，以免破坏自己的良好形象。☺）

在一些汇编编程论坛上，经常有初学者问到 MASM 和 TASM 有什么不同，用哪个比较好，@@@ 标号是什么意思，为什么用下载的汇编编译器无法编译程序等问题，虽然这些都属于最基本的问题，但是以前的确没有一个地方或者有一本书能系统全面地讲解这些问题。本书的**基础篇**就是因此而设，它们是：

- 第 1 章 背景知识
- 第 2 章 准备编程环境
- 第 3 章 使用 MASM

当搭建编译环境和对编译器的使用不再成为绊脚石的时候，初学者的问题往往集中在对 Windows 程序结构的迷惑上，消息驱动体系、窗口过程、与硬件隔绝的图形接口及资源文件等相对于 DOS 程序来说都是全新的内容。接下来的 4 章将深入讨论这些内容，通过这几章，读者应该开始习惯以 Windows 的方式考虑问题了（脑海中的 DOS 逐渐远去……），这就是本书的**初级篇**：

- 第 4 章 第一个窗口程序
- 第 5 章 使用资源
- 第 6 章 定时器和 Windows 时间
- 第 7 章 图形操作

Windows 系统不像 DOS 系统，它的应用程序界面是规范化的，统一的界面来自大量统一的界面控件，学习这些控件就等于学习如何编写 Windows 界面。下面的**界面篇**中的两章将探讨这方面的内容：

- 第 8 章 通用对话框
- 第 9 章 通用控件

学到这里为止，读者应该可以写出界面规范的标准的 Win32 程序了。但还是无法用这些程序来解决一些具体问题，因为有关 Windows 系统的高级特征的介绍还没有开始，如内

存管理、文件操作和多线程等。这些就是本书**系统篇**中将要介绍的内容，通过这些内容，读者将比较深入地了解 Windows 的工作方式：

- 第 10 章 内存管理和文件操作
- 第 11 章 动态链接库和钩子
- 第 12 章 多线程
- 第 13 章 进程控制
- 第 14 章 异常处理

相信到这里为止，读者对 Windows 的了解已经比较系统了。虽然 Windows 中还存在其他很多方面的内容，如管道，邮件槽，如何写控制面板程序、屏幕保护程序和驱动程序等。但是有了前面的基础以后，读者自己去了解这些内容就不成问题了，因为掌握了“渔”，得到“鱼”又有什么困难呢？在最后的几章中，本书将从应用的角度再补充介绍一些常用的网络编程、注册表、PE 文件和数据库操作方面的内容，这就是**应用篇**：

- 第 15 章 注册表和 INI 文件
- 第 16 章 WinSock 接口和网络编程
- 第 17 章 PE 文件
- 第 18 章 ODBC 数据库编程

在本书中，笔者特别以显著的方式标出了一些经验之谈，这些是笔者在长期的汇编编程中得到的体会，可能是任何一本教科书或者手册里都没有的。希望这些能给读者带来帮助！



用“灯泡”标出的部分表示一些小技巧，可以对编程的理解有促进作用。



用“惊叹号”标出的部分表示容易出错的部分，可以帮助读者避免一些难以理解的错误。

对读者的假设

有了内容的定位，读者的定位也就比较清楚了，本书适合于以下读者：

- 想用 Win32 汇编写 Windows 应用程序的读者。
- 想从 DOS 下的 16 位汇编转向 Windows 下 32 位汇编的读者。
- 欲了解 Win32 汇编，以便为 Windows 下的加密解密、系统安全、逆向工程等方面打基础的读者。
- 欲了解 Win32 汇编，以便为用汇编写 Windows 驱动程序打基础的读者。
- 正在学习汇编课程，需要补充汇编课程中 Win32 部分的学生。

在开始本书之前，读者应该有以下的基础知识：

- 计算机的基础知识，如进制转换、逻辑运算、变量类型和指针的概念等。
- 数据结构的基础知识，因为 Win32 编程涉及大量的数据结构。
- C 语言的基础知识，因为 Win32 编程的绝大部分参考资料都是以 C 的格式出现的。
- Intel 80x86 处理器的基础知识，如寻址方式和指令的使用等。

本书并不是为以下读者准备的：

- 欲详细了解保护模式的读者——因为 Windows 并不是一个开放的平台，Windows 的开放只限于应用程序接口，所以要用 Windows 做背景研究保护模式只能是自讨苦吃，如果读者需要深入了解这方面的内容，最好的方法就是去研究 Linux 的核心代码并在 Linux 上实验。
- 欲了解 Windows 核心“机密”的读者——汇编并不等同于深入操作系统的内部，所以本书不是《Windows 内核分析》。而真正意义上的《Windows 内核分析》除了 Microsoft，恐怕谁也写不出来。
- 欲了解 Windows 驱动程序编写的读者——要介绍清楚 Windows 驱动程序，需要的篇幅绝不会亚于本书的篇幅，本书不打算涉及这方面的内容，读者有兴趣的话，可以阅读《Programming WDM》和《System Programming for Windows 95》等书，前者讲述的是 Windows 2000/Windows NT 下的 WDM 驱动程序，后者讲述的是 Windows 9x 下的 VxD 驱动程序。

第 3 版有什么新的内容

本书第 1 版出版至今已经 6 年多了，第 2 版出版至今也已经 3 年了，期间笔者收到了大量的读者来信，对本书提出了各种意见和建议，综合各种方面的考虑，本书的第 3 版做了以下改进。

- 对第 2 版中已知的错误进行了修正，包括一些排版错误、错别字和例子中的 Bug。
- 对一些过时的内容进行了更新或者删除。
- 根据读者的反馈，对部分章节进行了重写。

关于附书代码和读者反馈

为了更好地说明 Win32 汇编的编程方法，本书附带了 70 多个例子，这些例子的源代码全部可以在附书光盘中找到，代码全部采用 MASM 格式编写，推荐使用的编译软件为 MASM32 SDK 软件包。

MASM32 SDK 软件包可以在以下地址下载：

MASM32 SDK 官方站点：<http://www.masm32.com>

作者的 Win32 汇编编程站点：<http://www.win32asm.com.cn>

本书中的例子代码已经经过了严格的防病毒测试，绝对不含任何病毒，但第 11 章的例子涉及钩子技术，第 17 章的例子涉及对 PE 文件进行操作，其中的小段代码与一些木马和病毒的特征码类似，以至于被一些杀毒软件误认为有未知病毒，请读者放心使用，不必顾虑。

虽然本书中所有的例子代码都已经在 Windows 98、Windows 2000、Windows XP 和 Windows Vista 下测试通过，但也有存在 Bug 的可能，如果发现代码存在错误或者发现书中有其他问题，请告知作者，以便在下一个版本中改进。如果读者有任何的反馈意见——不管是批评还是鼓励，都请和作者联系，作者的 E-mail 是 asm@zj165.com。如果发现 E-mail 地址无效，请访问作者网站 <http://www.win32asm.com.cn> 获取最新有效的 E-mail 地址。



致 谢

首先感谢我的父母亲，如果没有你们从小到大对我的培养，就没有这一切。也感谢我的妹妹，在很多关键的时候，你总是给予我很多的帮助。

感谢我的妻子小猪猪，在本书创作的时候，没有你的理解和支持，我不可能完成这样一部作品；在本书发行后的日子里，要不是你将逛街、买衣服、旅游的时间慷慨地贡献出来，并盯紧四处乱跑的小宝宝，本书就不会有多次再版的机会。

感谢我的母校浙江大学，浙大“求是创新”的校训，“实事求是、严谨踏实、奋发进取、开拓创新”的校风让我能够有一个好的学习习惯，让我在毕业以后的这么多年里能够始终有一种动力去学习最新的知识。

感谢电子工业出版社博文视点资讯有限公司的郭立老师和李冰编辑，你们的支持、鼓励和专业的指点使本书能够按照进度按时完成。

非常高兴看到本书的再版，本书的前2个版本取得了累计销量达到40000册的好成绩，从本书的第1版发行到现在，我收到了很多读者和网友的反馈，使本书更加完善。在第3版发行之际，再次感谢你们对我的关心和爱护，也感谢你们为我提了很多宝贵的意见和建议。



目 录

基础篇

第 1 章 背景知识	1	2.5.1 Windows 资料的来源	40
1.1 Win32 的软硬件平台	1	2.5.2 Intel 处理器资料	42
1.1.1 80x86 系列处理器简史	1	2.6 构建编程环境	42
1.1.2 Windows 的历史	3	2.6.1 IDE 还是命令行	42
1.1.3 Win32 平台的背后—— Wintel 联盟	5	2.6.2 本书推荐的工作环境	43
1.2 Windows 的特色	6	2.6.3 尝试编译第一个程序	44
1.3 必须了解的基础知识	7	第 3 章 使用 MASM	46
1.3.1 80x86 处理器的工作模式	7	3.1 Win32 汇编源程序的结构	46
1.3.2 Windows 的内存管理	9	3.1.1 模式定义	48
1.3.3 Windows 的特权保护	17	3.1.2 段的定义	50
第 2 章 准备编程环境	21	3.1.3 程序结束和程序入口	53
2.1 Win32 可执行文件的 开发过程	21	3.1.4 注释和换行	53
2.2 编译器和链接器	23	3.2 调用 API	54
2.2.1 MASM 系列	23	3.2.1 API 是什么	54
2.2.2 TASM 系列	27	3.2.2 调用 API	56
2.2.3 其他编译器	28	3.2.3 API 参数中的等值定义	60
2.2.4 MASM, TASM 还是 NASM	29	3.3 标号、变量和数据结构	62
2.2.5 我们的选择——MASM32 SDK 软件包	30	3.3.1 标号	62
2.3 创建资源	31	3.3.2 全局变量	64
2.3.1 资源编译器的使用	31	3.3.3 局部变量	65
2.3.2 所见即所得的资源编辑器	32	3.3.4 数据结构	68
2.4 make 工具的用法	34	3.3.5 变量的使用	70
2.4.1 make 工具是什么	34	3.4 使用子程序	74
2.4.2 nmake 的用法	35	3.4.1 子程序的定义	75
2.4.3 描述文件的语法	36	3.4.2 参数传递和堆栈平衡	76
2.5 获取资料	39	3.5 高级语法	79
		3.5.1 条件测试语句	79
		3.5.2 分支语句	80
		3.5.3 循环语句	82
		3.6 代码风格	85
		3.6.1 变量和函数的命名	85

3.6.2	代码的书写格式	87	5.5	字符串资源	174
3.6.3	代码的组织	88	5.6	版本信息资源	176
			5.6.1	版本信息资源的定义	176
			5.6.2	在程序中检测版本信息	179
			5.7	二进制资源和自定义资源	180
			5.7.1	使用二进制资源	180
			5.7.2	使用自定义资源	181
初级篇					
第 4 章	第一个窗口程序	89	第 6 章	定时器和 Windows 时间	183
4.1	开始了解窗口	89	6.1	定时器	183
4.1.1	窗口是什么	89	6.1.1	定时器简介	183
4.1.2	窗口界面	90	6.1.2	定时器的使用方法	184
4.1.3	窗口程序是怎么工作的	92	6.2	Windows 时间	188
4.2	分析窗口程序	98	6.2.1	Windows 时间的获取和设置	188
4.2.1	模块和句柄	98	6.2.2	计算时间间隔	189
4.2.2	创建窗口	100	第 7 章	图形操作	191
4.2.3	消息循环	107	7.1	GDI 原理	191
4.2.4	窗口过程	109	7.1.1	GDI 程序的结构	192
4.3	窗口间的通信	114	7.1.2	设备环境	195
4.3.1	窗口间的消息互发	114	7.1.3	色彩和坐标	201
4.3.2	在窗口间传递数据	117	7.2	绘制图形	203
4.3.3	SendMessage 和 PostMessage 函数的区别	118	7.2.1	画笔和画刷	211
第 5 章	使用资源	119	7.2.2	绘制像素点	214
5.1	菜单和加速键	120	7.2.3	绘制图形	214
5.1.1	菜单和加速键的组成	120	7.2.4	绘图模式	218
5.1.2	菜单和加速键的资源定义	121	7.3	创建和使用位图	220
5.1.3	使用菜单和加速键	126	7.3.1	一个使用位图的时钟例子	220
5.2	图标和光标	138	7.3.2	创建和使用位图	230
5.2.1	图标和光标的资源定义	139	7.3.3	使用设备无关位图	231
5.2.2	使用图标和光标	139	7.4	块传送操作	233
5.3	位图	143	7.4.1	块传送方式	233
5.3.1	位图简介	143	7.4.2	块传送函数	234
5.3.2	在资源中定义位图	144	7.5	区域和路径	239
5.4	对话框	145	7.5.1	使用区域	239
5.4.1	对话框简介	145	7.5.2	使用路径	241
5.4.2	对话框的资源定义	147			
5.4.3	使用对话框	149			
5.4.4	在对话框中使用子窗口控件	152			

界面篇

第 8 章 通用对话框	243
8.1 通用对话框简介	243
8.2 使用通用对话框	250
8.2.1 “打开”文件和“保存” 文件对话框	250
8.2.2 字体选择对话框	252
8.2.3 “颜色选择”对话框	254
8.2.4 “查找”和“替换”文本 对话框	255
8.2.5 “页面设置”对话框	258
8.2.6 “浏览目录”对话框	259
第 9 章 通用控件	260
9.1 通用控件简介	260
9.1.1 通用控件的分类	260
9.1.2 使用通用控件	262
9.2 使用状态栏	266
9.2.1 创建状态栏	271
9.2.2 状态栏的控制消息	272
9.2.3 在状态栏上显示菜单 提示信息	274
9.3 使用工具栏	275
9.3.1 创建工具栏	282
9.3.2 工具栏的控制消息	285
9.3.3 工具栏的通知消息	288
9.4 使用 Richedit 控件	291
9.4.1 创建 Richedit 控件	303
9.4.2 Richedit 控件的控制消息	305
9.4.3 Richedit 控件的通知消息	314
9.5 窗口的子类化	315
9.5.1 什么是窗口的子类化	315
9.5.2 窗口子类化的实现	316
9.6 控件的超类化	322
9.6.1 什么是控件的超类化	322
9.6.2 控件超类化的实现	323

系统篇

第 10 章 内存管理和文件操作	327
10.1 内存管理	327
10.1.1 内存管理基础	327
10.1.2 内存的当前状态	328
10.1.3 标准内存管理函数	330
10.1.4 堆管理函数	335
10.1.5 虚拟内存管理函数	339
10.1.6 其他内存管理函数	344
10.2 文件操作	345
10.2.1 Windows 的文件 I/O	345
10.2.2 创建和读写文件	346
10.2.3 查找文件	357
10.2.4 文件属性	364
10.2.5 其他文件操作	366
10.3 驱动器和目录	367
10.3.1 逻辑驱动器操作	368
10.3.2 目录操作	371
10.4 内存映射文件	373
10.4.1 内存映射文件简介	374
10.4.2 使用内存映射文件	376
第 11 章 动态链接库和钩子	384
11.1 动态链接库	384
11.1.1 动态链接库的概念	384
11.1.2 编写动态链接库	385
11.1.3 使用动态链接库	391
11.1.4 动态链接库中的数据 共享	400
11.1.5 在 VC++ 中使用动态 链接库	401
11.2 Windows 钩子	404
11.2.1 什么是 Windows 钩子	404
11.2.2 远程钩子的安装和使用	406
11.2.3 日志记录钩子	414

第 12 章 多线程	418	14.3 使用 SEH 处理异常	510
12.1 进程和线程	418	14.3.1 注册回调函数	512
12.2 多线程编程	419	14.3.2 异常处理回调函数	513
12.2.1 一个单线程的“问题 程序”	419	14.3.3 SEH 链和异常的传递	516
12.2.2 多线程的解决方法	423	14.3.4 展开操作 (Unwinding) ...	518
12.2.3 与线程有关的函数	427		
12.3 使用事件对象控制线程	431	应用篇	
12.3.1 事件	432	第 15 章 注册表和 INI 文件	522
12.3.2 等待事件	433	15.1 注册表和 INI 文件简介	522
12.3.3 进一步改进计数程序	434	15.2 INI 文件的操作	523
12.4 线程间的同步	437	15.2.1 INI 文件的结构	523
12.4.1 产生同步问题的原因	437	15.2.2 管理键值	525
12.4.2 各种用于线程间同步 的对象	442	15.2.3 管理小节	532
		15.2.4 使用不同的 INI 文件	533
第 13 章 过程控制	450	15.3 对注册表的操作	534
13.1 环境变量和命令行参数	450	15.3.1 注册表的结构	534
13.1.1 环境变量	450	15.3.2 管理子键	536
13.1.2 命令行参数	453	15.3.3 管理键值	547
13.2 执行可执行文件	458	15.3.4 子键和键值的枚举	548
13.2.1 方法一: Shell 调用	458	15.3.5 注册表应用举例	551
13.2.2 方法二: 创建进程	460		
13.3 进程调试	469	第 16 章 WinSock 接口和网络编程	555
13.3.1 获取运行中的进程句柄	469	16.1 Windows Socket 接口简介	556
13.3.2 读写进程的地址空间	475	16.2 Windows Socket 接口的 使用	559
13.3.3 调试 API 的使用	480	16.2.1 IP 地址的转换	559
13.4 进程的隐藏	489	16.2.2 套接字	563
13.4.1 在 Windows 9x 中隐藏 进程	489	16.2.3 网络应用程序的一般 工作流程	566
13.4.2 Windows NT 中的远程 线程	491	16.2.4 监听、发起连接和 接收连接	569
第 14 章 异常处理	503	16.2.5 数据的收发	572
14.1 异常处理的用途	503	16.2.6 一个最简单的 TCP 服务 端程序	575
14.2 使用筛选器处理异常	504	16.3 TCP 应用程序的设计	581
14.2.1 注册回调函数	504	16.3.1 通信协议和工作线程 的设计	581
14.2.2 异常处理回调函数	506		

16.3.2 TCP 聊天室例子——服务器端	591	17.5 重定位表	667
16.3.3 TCP 聊天室例子——客户端	598	17.5.1 重定位表的结构	667
16.3.4 以非阻塞方式工作的 TCP 聊天室客户端	606	17.5.2 查看 PE 文件的重定位表举例	670
16.3.5 其他常用函数	617	17.6 应用实例	672
第 17 章 PE 文件	621	17.6.1 动态获取 API 入口地址	672
17.1 PE 文件的结构	621	17.6.2 在 PE 文件上添加执行代码	679
17.1.1 概论	621	第 18 章 ODBC 数据库编程	688
17.1.2 DOS 文件头和 DOS 块	622	18.1 基础知识	688
17.1.3 PE 文件头 (NT 文件头)	624	18.1.1 数据库接口的发展历史	688
17.1.4 节表和节	629	18.1.2 SQL 语言	691
17.2 导入表	644	18.1.3 ODBC 程序的流程	693
17.2.1 导入表简介	644	18.2 连接数据库	694
17.2.2 导入表的结构	646	18.2.1 连接和断开数据库	694
17.2.3 查看 PE 文件导入表举例	649	18.2.2 连接字符串	700
17.3 导出表	651	18.3 数据的管理	703
17.3.1 导出表的结构	652	18.3.1 执行 SQL 语句	703
17.3.2 查看 PE 文件导出表举例	655	18.3.2 执行结果的处理	708
17.4 资源	658	18.3.3 获取结果集中的数据	710
17.4.1 资源简介	658	18.3.4 事务处理	715
17.4.2 资源的组织方式	659	18.4 数据库操作的例子	717
17.4.3 查看 PE 文件中的资源列表举例	662	18.4.1 结果集处理模块	718
		18.4.2 例子的源代码	723
		参考文献	734

第 1 章

背景知识

让我们在轻松的背景知识介绍中开始 Win32 汇编之旅。本章将对 Win32 平台的历史和现状做简要介绍，同时对 80386 处理器，以及 Windows 操作系统中涉及 Win32 汇编的基础知识部分做快速充电。

1.1 Win32 的软硬件平台

1.1.1 80x86 系列处理器简史

Win32 可以在多种硬件平台上运行，但使用最广泛的硬件平台是基于 Intel 公司 80x86 系列处理器的微型计算机。

自 1978 年 6 月 Intel 公司推出它的第一个 16 位微处理器 8086 以来，计算机技术就开始进入飞速发展的时期。8086 芯片的主频为 4.43 MHz，集成的晶体管数大约为 2.9 万个，运算器的位长为 16 位，采用了 20 条地址线，可以寻址的范围为 2^{20} 个字节地址，即 1 MB；1982 年，该公司发布了 80286 处理器，芯片上集成了 12 万个晶体管，主频提高到了 12 MHz。

1985 年 Intel 公司推出 32 位的 80386 处理器，芯片上集成的晶体管数为 27.5 万个，主频提高到了 33 MHz，地址线则扩展到 32 条，直接寻址的能力达到 4 GB。80386 处理器在设计的时候考虑了多用户及多任务的需要，在芯片中增加了保护模式、优先级、任务切换和片内的存储单元管理等硬件单元。80386 的出现使 Windows 和 UNIX 等多任务的操作系统可以在 PC 上运行。直到现在，运行于 80x86 处理器之上的多任务操作系统都是以 80386 的运行模式为基础的。

1989 年，Intel 公司推出 80486 处理器，在芯片内集成了浮点处理器和 8 KB 的一级缓存，片内的晶体管数达到了 118 万个，并把主频提高到 50 MHz~66 MHz。80486 处理器开始使用流水线技术，即在 CPU 中由 5~6 个不同功能的电路单元组成一条指令处理流水线，然后将一条指令分成 5~6 步后再由这些电路单元分别执行，由此提高 CPU 的运算速度。电路单元的数目就是流水线的深度。为了使计算机中的其他部件不至于成为 CPU 速

度发展的瓶颈，80486 处理器开始使用了倍频技术，即让处理器速度（CPU 主频）数倍于系统总线速度（外频）。

从 80386 开始，在 Intel 公司向市场大量推出处理器芯片的同时，其他一些电脑公司和厂商如 AMD 和 Cyrix 等，也纷纷投入大量的人力财力进行处理器的开发和研制，并很快把研制出的产品推向市场。这些 CPU 芯片和 80386 芯片兼容，在编程上可以使用与 Intel 处理器相同的指令集。

1993 年 3 月 Intel 公司推出 80586 处理器。由于无法阻止其他公司把自己的兼容产品也叫做 x86，所以把产品取名为 Pentium，并且进行了商标注册，同时启用了中文名称“奔腾”。Pentium 芯片中集成了 310 万个晶体管，内置 16 KB 缓存，主频有 60 MHz 和 66 MHz 两个版本，后来逐步提高，到 1995 年 6 月时主频提高到了 133 MHz。Pentium 处理器采用许多新技术，其中最重要的变化是采用了超标量体系结构。即将两个同时工作的指令执行部件封装在同一芯片中，用两条并行的通道来执行指令，这相当于两个 CPU 同时工作，大大提高了处理速度。在 586 时代，AMD 和 Cyrix 等其他公司也推出了相应档次的 CPU，命名为 5x86 和 K5 等。

1995 年 11 月，Intel 公司推出代号为 P6 的新一代 Pentium Pro 处理器，中文名称为“高能奔腾”。Pentium Pro 芯片中集成了 550 万个晶体管，主频分 150 MHz~200 MHz 多个版本。片内集成了 3 条平行的指令执行通道，相当于 3 个 CPU 并行工作，并用超流水线技术将流水线的深度提高到了 14 级。P6 处理器内置 16 KB 一级高速缓存，并将 256 KB 或 512 KB 的二级高速缓存芯片与 CPU 内核芯片同时封装在一个外壳中，缩短了 CPU 和二级高速缓存之间的线路走线距离。同时，P6 处理器开始使用乱序执行和分支预测技术，这使下一条指令不一定要等到前一条指令执行完毕后才可以开始。所有这些技术使这种 CPU 在运行 32 位指令系统时的执行效率明显高于上一代 Pentium。

随着 CPU 和操作系统的发展，多媒体技术开始流行，依靠浮点处理器已经不能满足多媒体音频和视频信号的实时处理任务了。1997 年年初，Intel 公司在 Pentium Pro 芯片上增加了专用于多媒体处理的 57 条指令和 8 个 64 位专用寄存器，命名为 Pentium MMX。Pentium MMX 使用了 450 万个晶体管，最高主频达到了 233 MHz。

1997 年 5 月，Intel 公司又向市场推出了 Pentium II 芯片，中文名称为“奔腾 II 代”。Pentium II 内集成了 750 万个晶体管，最高主频达到了 300 MHz，也具有 MMX 的功能。这种处理器将二级高速缓存移到芯片外，以提高芯片成品率。为了照顾低端市场，1998 年 Intel 公司推出了除去二级高速缓存的 Pentium II 简化版，命名为 Celeron 处理器。由于其缺乏片内二级高速缓存，对速度的影响非常巨大，使 Celeron 处理器的实际性能非常低。1998 年 4 月，Intel 公司又把 128 KB 二级高速缓存加回到 Celeron 处理器中，命名为 Celeron A 处理器。Celeron A 的主频从 300 MHz 开始。

1999 年，Intel 公司推出集成了 950 万个晶体管，主频为 450 MHz~500 MHz，外频为 100 MHz 的 Pentium III 处理器。这种处理器新增了 SSE 指令集，提供 70 条全新的指令，可以大大提高 3D 运算、动画片、影像与音效等功能，增强了视频处理和语音识别的功能。

这套指令集主要为浏览 WWW 网页而设计。Pentium III 处理器在芯片内集成了 64 KB 的一级缓存，并将 512 KB 的二级缓存安装在外壳卡盒内。

2000 年 11 月，Intel 公司发布集成 4200 万个晶体管的 Pentium 4 处理器，主频达到了 1.4 GHz，系统总线速度为 400 MHz，流水线的深度提高到 20 级，并增加了 SSE2 指令集，提供 144 条新指令用于提高摄像、多媒体、3D 图像和密集运算等方面的速度。到 2005 年，Pentium 4 处理器的主频已经提升到了 3 GHz 以上。

在如此高的主频下，电压和发热量成为继续提高主频的主要障碍，导致无法再通过简单提升时钟频率来提升 CPU 性能。面对主频之路走到尽头，Intel 开始寻找其他方式来提升处理器的性能，而最具实际意义的创新是增加 CPU 内处理核心的数量。多核时代开创于 2005 年春季，其标志是 Intel 的 Pentium D 双核芯片，2006 年 1 月 Intel 发布了首款双核移动处理器 Core Duo，时至今日，四核技术的 CPU 也已经发布。

从第一块微处理器诞生至今，处理器技术发展出不少新的体系结构。从微处理器的指令系统来看，有两种分支走向，一种是 CISC；一种是 RISC。CISC 即复杂指令系统计算机。从 PC 诞生以来，人们一直沿用 CISC 指令集方式。它的指令不等长，指令的条数比较多，编程和设计处理器时都较为麻烦。在 CISC 之后，人们发明了 RISC，即精简指令系统。这种指令系统采用等长的指令，且指令数较少，通过简化指令可以让计算机的结构更为简单，进而提高运算速度。

Intel 的 80x86 系列处理器看起来属于 CISC 体系，但实际上，从 Pentium 处理器开始，都已不是单纯的 CISC 体系了。因为它们引入了很多 RISC 体系里的先进技术来大幅度提高性能。但是，好马也得配好鞍——没有软件支持的 CPU 再快也不是好 CPU。为了兼容已有的软件，80x86 系列处理器也不得不背上沉重的历史包袱。如 CPU 的位长还是停留在 32 位；在寄存器、运行模式与内存管理模式等方面还是继承了早期的 80386 模式；80386 以后的处理器虽然增加了不少新指令，但大多用于多媒体扩展，其中很少有和操作系统密切相关的指令。所以，如果不涉及 3D 及密集运算方面的运算，仅从操作系统的角度看，这些处理器只能算是一个快速的 80386 处理器而已。

1.1.2 Windows 的历史

Win32 指的是 32 位的 Windows 系操作系统。Microsoft 公司有一系列的 Windows 操作系统，下面先简单介绍 Windows 的历史。

谈到 Windows 的历史就不能不谈 MS-DOS 的历史。MS-DOS 的技术源自 CP/M 操作系统。1973 年，第一个 8 位磁盘操作系统 CP/M 出现，这种操作系统有较好的层次结构，它利用 BIOS 隔离硬件和操作系统的其他模块，有很好的可移植性和易用性。在此基础上，西雅图计算机公司于 1978 年开始开发 QDOS，此后又成功研制出 16 位微型机的实验性操作系统 86-DOS。

也正是在这段时期，IBM 公司正在开发基于 8086 处理器的 IBM PC，急需一个配套的操作系统，但和 CP/M 开发者之间的谈判不是很顺利，这时 Microsoft 公司乘虚而入。

Microsoft 没有足够的时间开发新的操作系统，于是找到了西雅图计算机公司，双方达成了由 Microsoft 经销 86-DOS 操作系统的协议。以 86-DOS 操作系统为基础，Microsoft 很快开发出 MS-DOS 1.0 版本。1981 年 8 月，MS-DOS 1.0 和 IBM PC 一起发布。

MS-DOS 1.0 还不支持硬盘和分层目录结构，文件管理中继承了 CP/M 操作系统的许多功能，但仅支持单面软盘。到了 1983 年，为了支持带硬盘的 PC/XT 计算机，经过较大地改造并吸取了 UNIX 的很多优点后，MS-DOS 升级到 2.0 版本，可以支持 32 MB 大小的硬盘分区。1984 年，MS-DOS 升级到 3.0 版本，开始支持 1.2 MB 软盘，用于 PC/AT 计算机。1986 年，为了支持 3.5 英寸软盘，MS-DOS 升级到 3.2 版本。

1987 年，为了兼容 IBM 和 PS/2 个人计算机，MS-DOS 升级到 3.3 版，这也是最流行的 DOS 版本。1990 年，Microsoft 推出 MS-DOS 5.0，开始支持 2.88 MB 的软盘，并可以把部分系统代码放到高端内存运行，空出低端内存供应用程序使用，同时将磁盘单个分区的支持容量提高到了 2 GB。

一直到 MS-DOS 的最后版本 6.22 为止，绝大多数的 PC 上运行的就是这个字符界面的操作系统。当时要想玩转 DOS，必须有专业计算机知识，不然“Bad command or filename”之类的提示随处可见，对此一般用户还真会不知所措。所以，“虽然界面简陋却令人兴奋”的 Windows 1.0 于 1985 年 11 月正式发布时，还是为沉闷的屏幕带来了一丝清新，毕竟它使非专业的人员使用计算机变得容易。在增强了键盘和鼠标接口后，1987 年微软又推出了 Windows 2.0 版。由于当时的硬件和 DOS 功能的限制，Windows 并不实用，所以这两个版本并不成功。Windows 2.0 版发布不久，Intel 公司的 80386 处理器发布，Microsoft 推出使用 80386 处理器 V86 模式的 Windows 2.1，即 Windows/286。

在接下来的时间里，基于 Intel 80x86 微处理器的 IBM 兼容机已经快速普及，这给 Microsoft 开发新的 Windows 系统提供了发展空间和市场。Microsoft 公司对 Windows 的内存管理和图形界面做了重大改进，在 1990 年 5 月份推出了 Windows 3.0，可以支持 Intel 80286/386/486 微处理器的保护模式，并可以访问达 16 MB 的内存。Windows 3.0 一面世便在商业上取得了惊人的成功，从而一举奠定了 Microsoft 在操作系统上的垄断地位。1992 年 4 月，Microsoft 推出了更稳定的 Windows 3.1，可以支持 True Type 字体。Windows 3.1 是 16 位 Windows 中最流行的版本。

1993 年 5 月，Microsoft 发布了具备安全性和稳定性特征的 32 位操作系统 Windows NT 3.11，主要针对网络和服务器市场。“NT”代表“新技术”（New Technology）。NT 3.11 是 Windows 系列中使用 32 位编程模式的第一个版本。它充分利用 80386 及以上处理器的平坦地址空间和保护模式等新技术，并可以移植到 Alpha、MIPS 和 Power PC 等不同的处理器平台上运行。

随后，Microsoft 借 Windows 东风，于 1995 年 8 月推出新一代操作系统 Windows 95（又名 Chicago）。Windows 95 实现了很友好的用户界面，支持即插即用功能，支持主流多媒体设备和 DirectX 编程接口，成为 Microsoft 发展史上的一个里程碑，也是操作系统发展史上的一个里程碑。从此，Windows 9x 便取代了 Windows 3.x 和 MS-DOS 操作系统，成为

个人计算机平台的主流操作系统。

在 20 世纪 90 年代后期, Microsoft 根据家庭个人用户和商业办公用户的不同需求, 分别提供 Window 9x 和 Windows NT 这两个系列的操作系统, Windows 9x 注重用户界面及其他易用性特征, 而 NT 系列则在纯 32 位内核的稳定性和可靠性等企业级特征上下工夫; 另一方面, 特别针对不同规模商业用户的需求, Windows NT 系列分为工作站版和服务器版等多个版本。在 Windows 9x 系列上, 从 Windows 95 OSR2 版起, Microsoft 先后发布了 Windows 98, Windows 98 SE 和 Windows Me 这三个面向家庭和个人用户的 PC 操作系统; 而在商用操作系统领域, 继 Windows NT 3.11 之后, Microsoft 相继发布了 Windows NT 3.5 和 4.0 两代操作系统, 并在 NT 4.0 上采用了 Windows 95 式的用户界面。2000 年, Microsoft 发布采用纯 32 位内核并照顾了家庭消费类应用程序的 Windows NT 5.0, 即 Windows 2000。

为了利用 MS-DOS 时代大量的应用程序, 保持向下的兼容性, Windows 9x 的内核模块还有许多地方使用 16 位程序, 但在编程上支持 32 位的编程模式。Windows NT 系列和 Windows 9x 系列操作系统都支持 Win32 API (Application Programming Interface), 即 Windows 32 位应用程序编程接口, Win32 API 为应用程序提供了大量的系统功能调用, 通过 Win32 API 调用 Windows 系统相当于在 MS-DOS 中通过中断方式调用系统功能。就像 DOS 汇编程序中随处可见的 INT 21h 指令一样, Windows 应用程序中 Win32 API 也随处可见。

随着时代的发展, 针对个人用户领域, Microsoft 于 2001 年 10 月发布了 Windows XP, 2007 年 1 月份发布了 Windows Vista; 针对商业操作系统领域, 于 2003 年 5 月发布了 Windows Server 2003, 2008 年 2 月发布了 Windows Server 2008。虽然两个系列的操作系统侧重点各不相同, 个人操作系统侧重于文档管理、游戏、个人通讯、流媒体等功能, 商业操作系统侧重于活动目录、组策略和管理、磁盘管理等面向服务器的功能, 但是从编程的角度来讲, 这些操作系统并没有多少不同之处, 仍然使用 Win32 API 作为编程接口。

1.1.3 Win32 平台的背后——Wintel 联盟

Windows 是伴随着 Intel 80x86 系列处理器从弱小逐渐成为霸主的。在 20 世纪 90 年代, Intel 80x86 系列处理器更新换代最快的时期也就是 Microsoft 的 Windows 系列最红的时期。在这个时期, Windows 标志和 Intel Inside 标志几乎是每一台桌面 PC 上都有的烙印。Microsoft 和 Intel 公司一软一硬, 完全统治着全球 PC 机的市场, 成为整个 PC 时代的象征, 被世人称为 Wintel 联盟。

从 20 世纪 80 年代起, 当时规模甚小的 Microsoft 和 Intel 正式携手, 逐步垄断了计算机产业硬件与软件的主要领域。每当 Microsoft 推出功能更强的软件后, Intel 处理器的需求量就上升; 同样, 当 Intel 生产出速度更快的处理器后, Microsoft 的软件因有了更好的平台而显得更易用。Intel 有多快的 CPU, Microsoft 就有相应的、庞大的软件来支持它。Microsoft 的应用程序不管有多庞大, 需要多快的速度, Intel 的新处理器又总能满足它。业界也必须出奇一致地放弃原有的软硬件平台, 转到新平台上去。因为, 谁跟不上 Wintel 的步伐, 谁就极有可能被淘汰出局。

Wintel 联盟不仅是针对竞争对手的联盟，它还是迫使用户升级的同盟。在升级的循环中，多数用户往往为了一个应用而被迫升级整个系统。Intel 有多快的 CPU，Microsoft 就有多花哨、多庞大的操作系统与之相配合。操作系统的升级即意味着应用软件的全面升级，而应用软件的升级则意味着用户整个系统必须升级。如果用户还在原有的系统上工作，那么就再也得不到新软件的支持了，因为，所有的应用软件公司都不愿意在过时的操作系统上投资开发自己的应用软件。大家都有体会，运行 Windows 95 很快的 Pentium II/250 把 Windows 98 一装上去，立刻慢了下来，等到升级到了 Pentium III/450，Windows 98 运行起来很快了，Windows 2000 又出来了，“快速”的 PIII 又成了老牛。在又一轮的升级下，CPU 爬到了 1 GHz 以上，等到 Windows 2000 运行起来很舒畅了，再试一下 Windows XP，用户升级的欲望又出来了！结果，用户口袋里的银子永远不会有满的一天。

时钟走过 2000 年，Wintel 联盟已呈衰减之势。Sun，IBM，Oracle，Linux，垄断和司法部等名词让 Microsoft 感到头痛；AMD 的速龙和钻龙处理器也让 Intel 手忙脚乱。Microsoft 和 Intel 在利益上的冲突也越来越公开化，整个业界都感受到了 Wintel 联盟将土崩瓦解的气息。

不管业界风起云涌也好，一片死寂也好，Intel 80x86 平台和 Windows 是桌面计算机上最流行的配置已是不争的事实。为了自由和创新，我们可以去学习 Linux，但在更多的时候，学会 Win32 编程是不得已的选择，即使是全世界的计算机中只剩 50% 在运行 Windows，Windows 程序员仍然有广阔的用武之地，这也算是无奈之际给自己一个理由吧！

1.2 Windows 的特色

对于使用者来说，Windows 的特色毋需多言，下面的几个特点足以使它广泛流行：

- 图形用户界面（GUI，Graphic User Interface 的缩写词）——Windows 最重要的特色，用户由此摆脱了字符界面操作系统必须死记的键盘命令和令人一头雾水的屏幕提示，改为以鼠标为主可以直接和屏幕上所见即所得的界面进行交流。
- 一致的用户界面——使初学者便于使用，大部分的 Windows 程序界面看起来都是差不多的，都有菜单和标题栏等，掌握一个程序后就很容易尝试并掌握新的程序。
- 多任务——也是非常重要的特色，用户可以同时运行多个程序，一边工作一边听 MP3 显然是很吸引人的。另一个好处是用户可以在不同的程序之间传送数据。

但程序员更关心的是隐藏在底下的细节，Windows 究竟提供了什么便利？用 Win32 开发程序方便吗？对程序员来说，Windows 的以下特征更为重要：

- 大量的函数调用——Win32 支持上千种函数的调用，几乎涉及所有的方面，程序员可以把更多的时间放在程序的逻辑结构和用户界面上。
- 和设备的无关性——Win32 程序并不直接访问屏幕、打印机和键盘等硬件设备，Windows 虚拟了所有的硬件。只要有硬件的设备驱动程序，这个硬件就可以使用，应用程序并不需要关心硬件的具体型号。与 DOS 编程中需要针对不同的显示卡和打印机等编写很多的驱动程序来比，这个特性对程序员的帮助是巨大的。

- 内存管理——由于内存分页和虚拟内存的使用，每个程序都可以使用 4 GB 的地址空间，DOS 编程时必须考虑的 640 KB 内存问题已经成为历史。

1.3 必须了解的基础知识

1.3.1 80x86 处理器的工作模式

80386 处理器有 3 种工作模式：实模式、保护模式和虚拟 86 模式。实模式和虚拟 86 模式是为了和 8086 处理器兼容而设置的。在实模式下，80386 处理器就相当于一个快速的 8086 处理器。保护模式是 80386 处理器的主要工作模式。在此方式下，80386 可以寻址 4 GB 的地址空间，同时，保护模式提供了 80386 先进的多任务、内存分页管理和优先级保护等机制。为了在保护模式下继续提供和 8086 处理器的兼容，80386 又设计了一种虚拟 86 模式，以便可以在保护模式的多任务条件下，有的任务运行 32 位程序，有的任务运行 MS-DOS 程序。在虚拟 86 模式下，同样支持任务切换、内存分页管理和优先级，但内存的寻址方式和 8086 相同，也是可以寻址 1 MB 的空间。

由此可见，80386 处理器的 3 种工作模式各有特点且相互联系。实模式是 80386 处理器工作的基础，这时 80386 作为一个快速的 8086 处理器工作。在实模式下可以通过指令切换到保护模式，也可以从保护模式退回到实模式。虚拟 86 模式则以保护模式为基础，在保护模式和虚拟 86 模式之间可以互相切换，但不能从实模式直接进入虚拟 86 模式或从虚拟 86 模式直接退到实模式。

1. 实模式

80386 处理器被复位或加电的时候以实模式启动。这时候处理器中的各寄存器以实模式的初始化值工作。80386 处理器在实模式下的存储器寻址方式和 8086 是一样的，由段寄存器的内容乘以 16 当做基地址，加上段内的偏移地址形成最终的物理地址，这时候它的 32 位地址线只使用了低 20 位。在实模式下，80386 处理器不能对内存进行分页管理，所以指令寻址的地址就是内存中实际的物理地址。在实模式下，所有的段都是可以读、写和执行的。

实模式下 80386 不支持优先级，所有的指令相当于工作在特权级（优先级 0），所以它可以执行所有特权指令，包括读写控制寄存器 CR0 等。实际上，80386 就是通过实模式下初始化控制寄存器，GDTR, LDTR, IDTR 与 TR 等管理寄存器以及页表，然后再通过加载 CR0 使其中的保护模式使能位置位而进入保护模式的。实模式下不支持硬件上的多任务切换。

实模式下的中断处理方式和 8086 处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和 8086 处理器一样，每 4 个字节组成一个中断向量，其中包括两个字节的段地址和两个字节的偏移地址。

从编程的角度看，除了可以访问 80386 新增的一些寄存器外，实模式的 80386 处理器比 8086 有什么进步呢？其实最大的好处是可以使用 80386 的 32 位寄存器，用 32 位的寄存器进行编程可以使计算程序更加简捷，也加快了执行速度。比如在 8086 时代用 16 位寄

寄存器来完成 32 位的乘法和除法时，要进行的步骤实在是太多了，于是考试时出这一类的题目就成了老师们的最爱，所以那时候当学生的做梦都想着让寄存器的位数快快长，现在梦想终于成真了，用 32 位寄存器一条指令就可以完成（问题是老师们也发现了这个投机取巧的办法，为了达到让学生们基础扎实的目的，也把题目换成了 64 位的乘法和除法，不过有个好消息是 64 位处理器已经出现了）；其次，80386 中增加的两个辅助段寄存器 FS 和 GS 在实模式下也可以使用，这样，同时可以访问的段达到了 6 个而不必考虑重新装入的问题；最后，很多 80386 的新增指令也使一些原来不很方便的操作得以简化，如 80386 中可以使用下述指令进行数组访问：

```
mov    cx, [eax + ebx * 2 + 数组基地址]
```

这相当于把数组中下标为 `eax` 和 `ebx` 的项目放入 `cx` 中；`ebx * 2` 中的 2 可以是 1, 2, 4 或 8，这样就可以支持数组项的位长为 8 位到 64 位的数组。而在 8086 处理器中，实现相同的功能要进行一次乘法和两次加法。另外，`pushad` 和 `popad` 指令可以一次把所有 8 个通用寄存器的值压入或从堆栈中弹出，比起用下面的指令分别将 8 个寄存器入栈快了很多：

```
push    eax
push    ebx
...
pop     ebx
pop     eax
```

当然，使用了这些新指令的程序是无法拿回到 8086 处理器上去执行的，因为这些指令的编码在 8086 处理器上是未定义的。

2. 保护模式

当 80386 工作在保护模式下时，它的所有功能都是可用的。这时 80386 所有的 32 根地址线都可供寻址，物理寻址空间高达 4 GB。在保护模式下，支持内存分页机制，提供了对虚拟内存的良好支持。虽然与 8086 可寻址的 1 MB 物理地址空间相比，80386 可寻址的物理地址空间可谓很大，但实际的微机系统极少安装如此大的物理内存。所以，为了运行大型程序和真正实现多任务，虚拟内存是一种必需的技术。

保护模式下 80386 支持多任务，可以依靠硬件仅在一条指令中实现任务切换。任务环境的保护工作是由处理器自动完成的。在保护模式下，80386 处理器还支持优先级机制，不同的程序可以运行在不同的优先级上。优先级分 4 个级别（0 级~3 级），操作系统运行在最高的优先级 0 上，应用程序则运行在比较低的级别上；配合良好的检查机制后，既可以在任务间实现数据的安全共享也可以很好地隔离各个任务。从实模式切换到保护模式是通过修改控制寄存器 CR0 的控制位 PE（位 0）来实现的。在这之前还需要建立保护模式必需的一些数据表，如全局描述符表 GDT 和中断描述符表 IDT 等。

DOS 操作系统运行于实模式下，而 Windows 操作系统运行于保护模式下。

3. 虚拟 86 模式

虚拟 86 模式是为了在保护模式下执行 8086 程序而设置的。虽然 80386 处理器已经提

供了实模式来兼容 8086 程序，但这时 8086 程序实际上只是运行得快了一点，对 CPU 的资源还是独占的。在保护模式的多任务环境下运行这些程序时，它们中的很多指令和保护模式环境格格不入，如段寻址方式、对中断的处理和 I/O 操作的特权问题等。为了在保护模式下工作而丢弃这些程序的代价是巨大的。设想一下，如果 Windows 或 80386 处理器推出的时候宣布不能运行以前的 MS-DOS 程序，那么就等于放弃了一个巨大的软件库，Windows 以及 80386 处理器可能就会落得和苹果机一样的下场，这是 Microsoft 和 Intel 都不愿看到的。所以，80386 处理器又设计了一个虚拟 86 模式。

虚拟 86 模式是以任务形式在保护模式上执行的，在 80386 上可以同时支持由多个真正的 80386 任务和虚拟 86 模式构成的任务。在虚拟 86 模式下，80386 支持任务切换和内存分页。在 Windows 操作系统中，有一部分程序专门用来管理虚拟 86 模式的任務，称为虚拟 86 管理程序。

既然虚拟 86 模式以保护模式为基础，它的工作方式实际上是实模式和保护模式的混合。为了和 8086 程序的寻址方式兼容，虚拟 86 模式采用和 8086 一样的寻址方式，即用段寄存器乘以 16 当做基址再配合偏移地址形成线性地址，寻址空间为 1 MB。但显然多个虚拟 86 任务不能同时使用同一位置的 1 MB 地址空间，否则会引起冲突。操作系统利用分页机制将不同虚拟 86 任务的地址空间映射到不同的物理地址上去，这样每个虚拟 86 任务看起来都认为自己在使用 0~1 MB 的地址空间。

8086 代码中有相当一部分指令在保护模式下属于特权指令，如屏蔽中断的 cli 和中断返回指令 iret 等。这些指令在 8086 程序中是合法的。如果不让这些指令执行，8086 代码就无法工作。为了解决这个问题，虚拟 86 管理程序采用模拟的方法来完成这些指令。这些特权指令执行的时候引起了保护异常。虚拟 86 管理程序在异常处理程序中检查产生异常的指令，如果是中断指令，则从虚拟 86 任务的中断向量表中取出中断处理程序的入口地址，并将控制转移过去；如果是危及操作系统的指令，如 cli 等，则简单地忽略这些指令，在异常处理程序返回的时候直接返回到下一条指令。通过这些措施，8086 程序既可以正常地运行下去，在执行这些指令的时候又觉察不到已经被虚拟 86 管理程序做了手脚。MS-DOS 应用程序在 Windows 操作系统中就是这样工作的。

1.3.2 Windows 的内存管理

在这一节中，读者可以解决初学 Win32 汇编时的两大疑问：

- Win32 汇编中，每个程序都可以用 4 GB 的内存吗？
- Win32 汇编源代码中为什么看不到 CS，DS，ES 和 SS 等段寄存器的使用？

1. DOS 操作系统的内存安排

Win32 编程相对于 DOS 编程最大的区别之一就是内存的使用。

先来回顾一下 DOS 操作系统的内存使用，如图 1.1 所示。DOS 操作系统运行于实模式中，由于 8086 处理器的寻址范围只有 1 MB，当时系统硬件使用的存储器地址被安排在高

地址是从 A0000h（即 640 KB）开始的 384 KB 中，其中有用于显示的视频缓冲区和 BIOS 的地址空间。而在内存低端，安排了中断向量表和 BIOS 数据区；剩下从 500h 开始到 A0000h 总共不到 640 KB 的内存是操作系统和应用程序所能够使用的；应用程序不可能使用这 640 KB 以外的内存。这就是著名的“640 KB 限制”。而即使在这 640 KB 中，DOS 操作系统又占领了低端的一部分内存，最后剩下 600 KB 左右的内存才是应用程序真正可以用的。如果系统中有内存驻留程序存在，那么应用程序还要和这些 TSR 程序共同分享这段内存空间。

当 80386 处理器推出后，可以寻址的内存范围达到了 4 GB，利用 XMS 驱动程序可以访问到所有的 4 GB 地址空间。但 16 位的段寻址方式限制了 DOS 程序，“可见”的内存范围还是停留在 00000h 到 FFFF0h+64 KB 的范围内，所有高于 1 MB 的扩展内存只能通过 XMS 驱动程序当做数据交换使用，程序的执行空间并没有什么增加。



图 1.1 DOS 操作系统的内存安排

2. 80386 的内存寻址机制

Windows 的内存管理和 DOS 的内存管理有很大的不同，在了解 Windows 的内存管理模式之前，需要对 80386 保护模式下内存分页机制有所了解。为了做个对比，先来看实模式下的内存寻址方式，即 DOS 下的寻址方式，如图 1.2 所示。

在实模式下，一个完整的地址由段地址和偏移地址两部分组成。段地址放在 16 位的段寄存器中，然后在指令中用 16 位的偏移地址寻址。处理器换算时先将段地址乘以 10h，得到段在物理内存中的起始地址；然后加上 16 位的偏移地址得到实际的物理地址。如 xxxx:yyyy 格式的虚拟地址在内存中的实际位置是 $xxxx \times 10h + yyyy$ 。

当 80386 处理器工作在保护模式和虚拟 8086 模式的时候，可以使用全部 32 根地址线访问 4 GB 大的内存。段地址加偏移地址的计算方法显然无法覆盖这么大的范围。但计算一下就可以发现，实际上和 8086 同样的限制已经不复存在，因为 80386 所有的通用寄存器都是 32 位的， 2^{32} 相当于 4G，所以用任何一个通用寄存器来间接寻址，不必分段就已经可以访问到所有的内存地址。

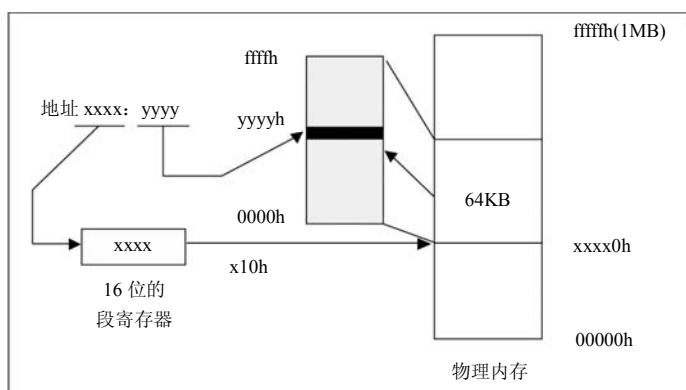


图 1.2 实模式下的内存寻址方式

这是不是说，在保护模式下，段寄存器就不再有用了呢？答案是否定的。实际上段寄存器更有用了，虽然在寻址上不再有分段的限制问题，但在保护模式下，一个地址空间是否可以被写入，可以被多少优先级的代码写入，是不是允许执行等涉及保护的问题就出来了。要解决这些问题，必须对一个地址空间定义一些安全上的属性。段寄存器这时就派上了用途。但是涉及属性和保护模式下段的其他参数，要表示的信息太多了，要用 64 位长的数据才能表示。我们把这 64 位的属性数据叫做段描述符（Segment Descriptor）。

80386 的段寄存器是 16 位的，无法放下保护模式下 64 位的段描述符。如何解决这个新的问题呢？解决办法是把所有段的段描述符顺序放在内存中的指定位置，组成一个段描述符表（Descriptor Table）；而段寄存器中的 16 位用来做索引信息，指定这个段的属性用段描述符表中的第几个描述符来表示。这时，段寄存器中的信息不再是段地址了，而是段选择器（Segment Selector）。可以通过它在段描述符表中“选择”一个项目以得到段的全部信息。

既然如此，段描述符表放在哪里呢？80386 中引入了两个新的寄存器来管理段描述符表。一个是 48 位的全局描述符表寄存器 GDTR，一个是 16 位的局部描述符表寄存器 LDTR。那么，为什么有两个描述符表寄存器呢？

GDTR 指向的描述符表为全局描述符表 GDT（Global Descriptor Table）。它包含系统中所有任务都可用的段描述符，通常包含描述操作系统所使用的代码段、数据段和堆栈段的描述符及各任务的 LDT 段等；全局描述符表只有一个。

LDTR 则指向局部描述符表 LDT（Local Descriptor Table）。80386 处理器设计成每个任务都有一个独立的 LDT。它包含有每个任务私有的代码段、数据段和堆栈段的描述符，也包含该任务所使用的一些门描述符，如任务门和调用门描述符等。

不同任务的局部描述符表分别组成不同的内存段，描述这些内存段的描述符当做系统描述符放在全局描述符表中。和 GDTR 直接指向内存地址不同，LDTR 和 CS，DS 等段选择器一样只存放索引值，指向局部描述符表内存段对应的描述符在全局描述符表中的位置。随着任务的切换，只要改变 LDTR 的值，系统当前的局部描述符表 LDT 也随之切换，

这样便于各任务之间数据的隔离。但 GDT 并不随着任务的切换而切换。

看到这里，读者可能会提出一个问题，既然有全局描述符表和局部描述符表两个表，那么段选择器中的索引值对应哪个表中的描述符呢。实际上，16 位的段选择器中只有高 13 位表示索引值。剩下的 3 个数据位中，第 0、1 位表示程序的当前优先级 RPL；第 2 位 TI 位用来表示在段描述符的位置；TI=0 表示在 GDT 中，TI=1 表示在 LDT 中。

以图 1.3 为例，在保护模式下，同样以 xxxx:yyyyyyyy 格式表示一个虚拟地址。单凭段选择器中的数值 xxxx 根本无法反映出段的基址在哪里。对于这个地址，首先要看 xxxx 的 TI 位是否为 0，如果是的话，则先从 GDTR 寄存器中获取 GDT 的基址（图中的步骤①），然后在 GDT 中以段选择器 xxxx 的高 13 位当做位置索引得到段描述符（步骤②）。段描述符包含段的基址、限长、优先级等各种属性，这就得到了段的起始地址（步骤③）；如果 xxxx 的 TI 位为 1 的话就更复杂了，这表示段描述符在 LDT 中，这时第一步的操作还是从 GDTR 寄存器中获取 GDT 的基址（步骤①'），并且要从 LDTR 中获取 LDT 所在段的位置索引（步骤②'）；然后以这个位置索引在 GDT 中得到 LDT 段的位置（步骤③'）；然后才是用 xxxx 做索引从 LDT 段中获得段描述符（步骤④'），再以这个段描述符得到段的基址等信息（步骤⑤'）。分这两种情况得到段的基址后（图中 Result 所示），再以基址加上偏移地址 yyyyyyyy 才得到最后的线性地址。

关于段描述符更详细的格式定义和使用说明，读者可以参考其他讲述保护模式的书籍。

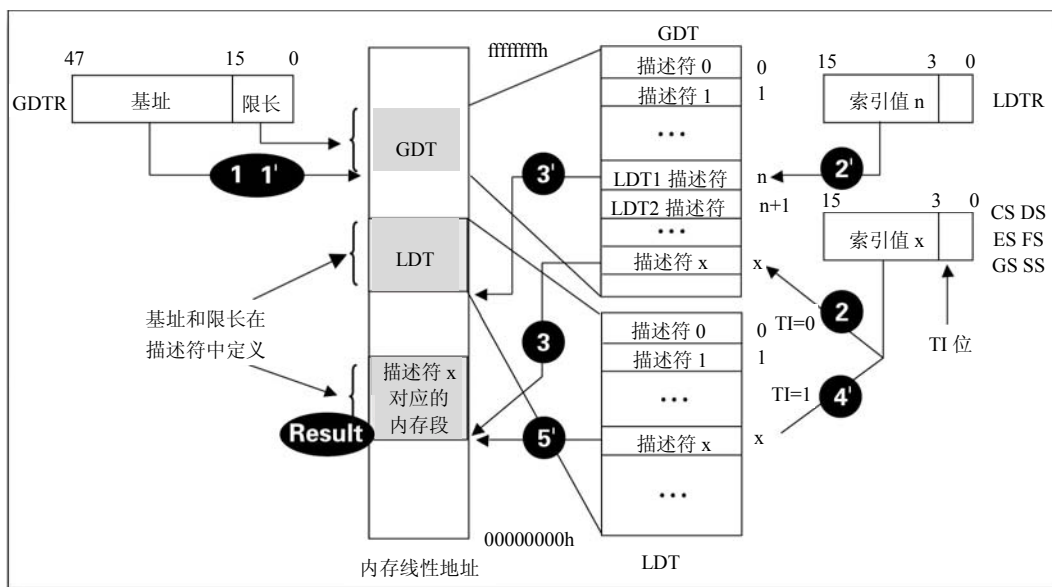


图 1.3 保护模式下 GDTR、LDTR、全局描述符表、局部描述符表和选择器的关系

3. 80386 的内存分页机制

读者可以注意到，在实模式下寻址的时候，“段寄存器+偏移地址”经过转换计算以

后得到的地址是“物理地址”，也就是在物理内存中的实际地址。而保护模式下，“段选择器+偏移地址”转换后的地址被称为“线性地址”而不是“物理地址”。那么，线性地址就是物理地址吗？

答案可能是“是”，也可能是“不是”，这取决于 80386 的内存分页机制是否被使用。

在单任务的 DOS 系统中，一个应用程序可以使用所有的空闲内存。程序退出后，操作系统回收所有的碎片内存并且合并成一个大块内存继续供下一个程序使用。内存合并过程中的一个极端情况是当系统中有多个 TSR 程序时，早装入内存的 TSR 被卸载后，后装入的 TSR 会留在内存的中间部位，把空闲内存隔成两个区域。这时应用程序使用的最大内存块只能是这两块内存中较大的一块，无法将它们合并使用。

对于一个多任务的操作系统，内存的碎片化是不能容忍的。否则，经过一段时间后，即使空闲内存的总和很大，也可能出现任何一片内存都小到无法装入执行程序的地步。所以多任务操作系统中碎片内存的合并是个很重要的问题。

80386 处理器的分页机制可以很好地解决这个问题。80386 处理器把 4 KB 大小的一块内存当做一“页”内存，每页物理内存可以根据“页目录”和“页表”，随意映射到不同的线性地址上。这样，就可以将物理地址不连续的内存的映射连到一起，在线性地址上视为连续。在 80386 处理器中，除了和 CR3 寄存器（指定当前页目录的地址）相关的指令使用的是物理地址外，其他所有指令都是用线性地址寻址的。

是否启用内存分页机制是由 80386 处理器新增的 CR0 寄存器中的位 31（PG 位）决定的。如果 PG=0，则分页机制不启用，这时所有指令寻址的地址（线性地址）就是系统中实际的物理地址；当 PG=1 的时候，80386 处理器进入内存分页管理模式，所有的线性地址要经过页表的映射才得到最后的物理地址。

以图 1.4 为例，一个 xxxx:yyyyyyyy 格式的虚拟地址，经过图 1.3 所示的段地址转换步骤后得到 32 位的线性地址 zzzzzzzz（步骤①）。当禁用分页机制时，线性地址就是物理地址，处理器直接从物理内存存取数据（步骤②）；当启用分页机制时，得到线性地址的方法还是一样（步骤①），但是还要根据页目录和页表指定的映射关系把地址映射到物理内存的真正位置上（步骤③）。然后，CPU 以映射后的物理地址在物理内存中存取数据。这个过程对于指令来说是透明的。

内存分页管理只能在保护模式下才可以实现，实模式不支持分页机制。但不管在哪种模式下，所有寻址指令使用的都是线性地址，程序不用关心数据最后究竟存放在物理内存的哪个地方。

页表规定的不仅是地址的映射，同时还规定了页的访问属性，如是否可写、可读和可执行等。比如把代码所在的内存页设置为可读与可执行，那么权限不够的代码向它写数据就会引发保护异常。利用这个机制可以在硬件层次上支持虚拟内存的实现。

如图 1.5 所示，页表可以指定一个页面并不真正映射到物理内存中。这样，访问这个页的指令会引发页异常错误。这时，处理器会自动转移到页异常处理程序中去。操作系统

可以在异常处理程序中将硬盘上的虚拟内存读到内存中并修改页表重新映射，然后重新执行引发异常的指令。这样指令可以正常执行下去。

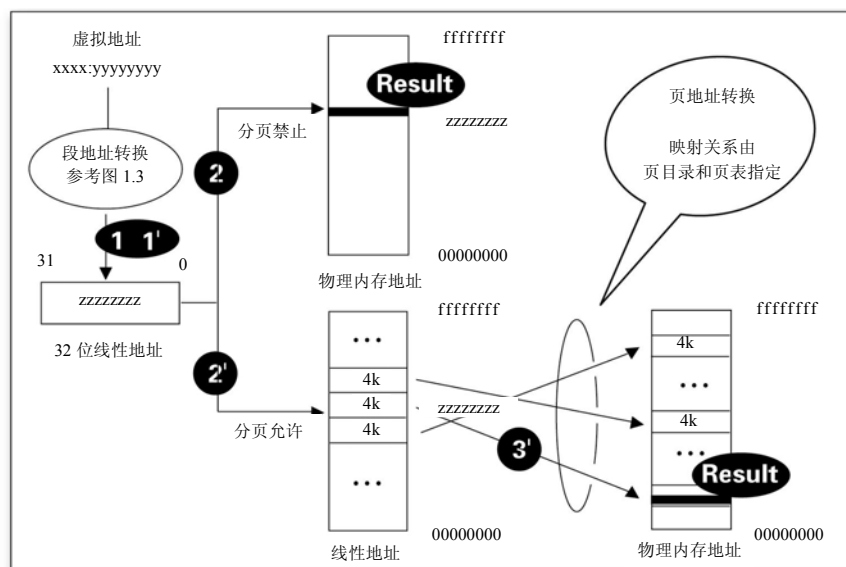


图 1.4 80386 的内存地址转换

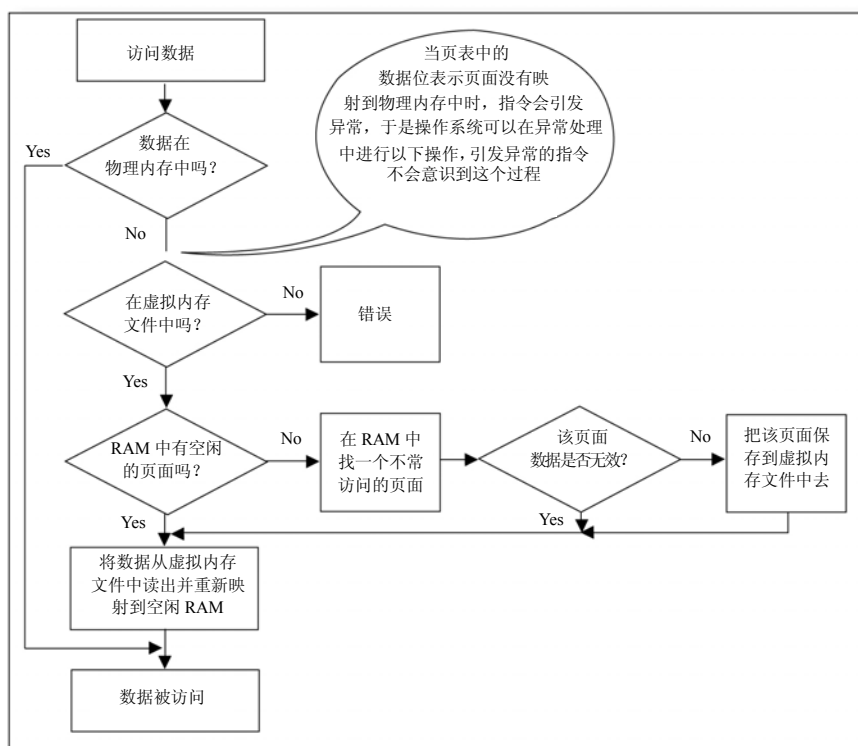


图 1.5 虚拟内存的实现

4. Windows 的内存安排

Windows 系统一般在硬盘上建立大小为物理内存两倍左右的交换文件（文件名在 Windows 9x 下为 Win386.swp，Windows NT 下为 PageFile.sys）用做虚拟内存。利用 80386 处理器的内存分页机制，交换文件在寻址上可以很方便地作为物理内存使用。只需在真正访问到的时候将硬盘文件的内容读入物理内存，然后重新将线性地址映射到这块物理内存就可以了。同样道理，被执行的可执行文件也不必真正装入内存，只要在页表中建立映射关系，以后真正运行到某处代码的时候再将它调入物理内存。

如果把虚拟内存暂时先视为物理内存的一部分，从物理内存的层次看，Windows 操作系统和 DOS 一样，也是所有的内容共享内存，比如操作系统使用的代码和数据（GDT，LDT 与页表等），当前执行中的所有程序的代码和数据以及这些程序调用的 DLL 的代码和数据等，如图 1.6 的左上角所示。

但是从应用程序代码的层次看，也就是说从分页映射后线性地址的层次看，内存的安排却不是这个样子。因为 Windows 是一个分时的多任务操作系统，CPU 时间被分成一个个的时间片后分配给不同程序轮流使用，在一个程序的时间片中，和这个程序执行无关的部分（如其他程序的代码和数据）并不需要映射到线性地址中去。

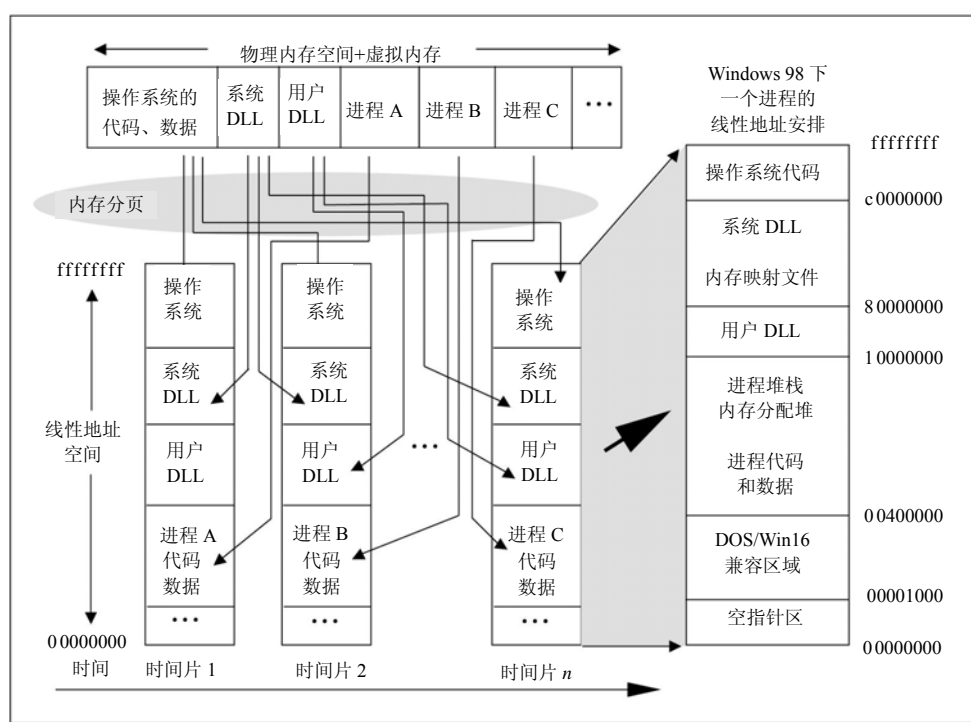


图 1.6 Windows 的内存安排

如图 1.6 所示，Windows 操作系统通过切换不同的页表内容让线性地址在不同的时间片中映射不同的内容。图中的右边是 Windows 98 操作系统在单个时间片中线性地址的安排。

排（Windows NT 的地址安排稍微有些不同）。在物理内存中，操作系统和系统 DLL 的代码需要供每个应用程序调用，所以在所有的时间片中都必须被映射；用户程序只在自己所属的时间片内被映射；而用户 DLL 则有选择地被映射。假设程序 A 和程序 C 都要用到 xxx.dll，那么物理内存中 xxx.dll 的代码在图中的时间片 1 和 n 中被映射，其他的时间片就不需要映射，当然，物理内存中只需要一份 xxx.dll 的代码。

由此可以引出 Win32 编程中几个很重要的概念：

- 每个应用程序都有自己的 4 GB 的寻址空间。该空间可存放操作系统、系统 DLL 和用户 DLL 的代码，它们之中有各种函数供应用程序调用。再除去其他的一些空间，余下的是应用程序的代码、数据和可以分配的地址空间。
- 不同应用程序的线性地址空间是隔离的。虽然它们在物理内存中同时存在，但在某个程序所属的时间片中，其他应用程序的代码和数据没有被映射到可寻址的线性地址中，所以是不可访问的。从编程的角度看，程序可以使用 4 GB 的寻址空间，而且这个空间是“私有”的。
- DLL 程序没有自己“私有”的空间。它们总是被映射到其他应用程序的地址空间中，当做其他应用程序的一部分运行。原因很简单，如果它不和其他程序同属一个地址空间，应用程序该如何调用它呢？

5. 从 Win32 汇编的角度看内存寻址

对初学者来说，DOS 下的分段寻址方式就已经令人一头雾水了，80386 保护模式的内存管理就更麻烦。的确，如果在 Win32 汇编中访问内存之前要先在描述符表中构造正确的描述符，然后再构造页表把物理内存映射到要访问的线性地址的话，那就简直是一场噩梦，有 90% 的汇编程序员会因此改行去卖茶叶蛋！

但实际上这并没有发生，因为 Win32 汇编中的内存访问远比 DOS 下的分段寻址方式简单，这是为什么呢？

因为 Windows 是一个多任务的操作系统，最首要的宗旨就是“稳定压倒一切”。如果把描述符表以及页表等内容交给用户程序管理是很不安全的，不用说全局描述符表，就是为每个程序建立的局部描述符表也不应该让用户程序改写，否则用户可以通过构造自己的描述符来访问操作系统不希望用户访问的代码或数据。任何权限上开放引发的安全问题都是很严重的，如 Windows 9x 中的中断描述符表是可写的，CIH 病毒可利用它将自己的权限提高到优先级 0；而 Windows NT 下的中断描述符表是不可写的，CIH 病毒在 Windows NT 下就无法使用同样的方法进驻内存。

正因为如此，Windows 操作系统干脆为用户程序“安排好了一切”。具体表现在为用户程序的代码段、数据段和堆栈段全部预定义好了段描述符。这些段的起始地址为 0，限长为 ffffffff，所以用它们可以直接寻址全部的 4 GB 地址空间。程序开始执行的时候，CS，DS，ES 和 SS 都已经指向了正确的描述符，在整个程序的生命周期内，程序员不必改动这些段寄存器，也不必关心它们的值究竟是多少（实际上是想改也改不了）。

所以对 Win32 汇编程序来说，整个源程序中竟然可以不用出现段寄存器的身影。这在 DOS 汇编编程中是不可想象的。回顾本节开头提出的问题，答案是：并不是 Win32 汇编源代码用不到段寄存器，而是用户在使用中不必去关心段寄存器！

1.3.3 Windows 的特权保护

Windows 的特权保护和处理器硬件的支持是分不开的。优先级的划分、指令的权限检查和超出权限访问的异常处理等是构成特权保护的基础。这一节将简单介绍这些课题，读者可以考虑一下初学 Win32 汇编时遇到的疑问：

- Win32 汇编中为什么找不到中断指令的使用？
- Windows 错误的“蓝屏幕”是从哪里来的？

1. 80386 的中断和异常

中断指当程序执行过程中有更重要的事情需要实时处理时（如串口中有数据到达，不及时处理数据会丢失，串行控制器就提交一个中断信号给处理器要求处理），硬件通过中断控制器通知处理器。处理器暂时挂起当前运行的程序，转移到中断处理程序中；当中断处理程序处理完毕后，通过 `iret` 指令回到原先被打断的程序中继续执行。

异常指指令执行中发生不可忽略的错误时（如遇到无效的指令编码，除法指令除零等），处理器用和中断处理相同的操作方法挂起当前运行的程序转移到异常处理程序中。异常处理程序决定在修正错误后是否回到原来的地方继续执行。

更为 DOS 汇编程序员熟悉的“中断”指的是用 `int n` 指令直接转移到中断向量 `n` 指定的中断处理程序中执行。严格地讲，`int n` 指令应该算“自陷”而不是“中断”。因为这时并不是程序被急需解决的事情打断，而是自己要求停止执行并转移到中断处理程序中去。

不管中断、异常还是自陷，虽然它们产生的原因不同，但处理过程是类似的，都通过中断向量表里存放的入口地址转移到服务程序，都由 CPU 自动在堆栈中保护断点地址，最后也都可以用 `iret` 指令返回指令被中断的地方。

先回顾一下 8086 或 80386 实模式下中断和异常的处理过程。如图 1.7 所示，实模式下的中断和异常服务程序地址存放在中断向量表中。中断向量表位于物理内存 `00000h` 开始的 `400h` 字节中，共支持 `100h` 个中断向量；每个中断向量是一个 `xxxx:yyyy` 格式的地址，占用 4 字节。当发生 `n` 号异常或 `n` 号中断，或者执行到 `int n` 指令的时候，CPU 首先到内存 `n×4` 的地方取出服务程序的地址 `aaaa:bbbb`（图示步骤①）；然后将标志寄存器、中断时的 CS 和 IP 压入堆栈，接着转移到 `aaaa:bbbb` 处执行（步骤②）；在服务程序最后遇到 `iret` 的时候，CPU 从堆栈中恢复标志寄存器，然后取出 CS 和 IP 并返回。

在保护模式下，中断或异常处理往往从用户代码切换到操作系统代码中执行。由于保护模式下的代码有优先级之分，因此出现了从优先级低的应用程序转移到优先级高的系统代码中的问题，如果优先级低的代码能够任意调用优先级高的代码，就相当于拥有了高优先级代码的权限。为了使高优先级的代码能够安全地被低优先级的代码调用，保护模式下

增加了“门”的概念。“门”指向某个优先级高的程序所规定的入口点，所有优先级低的程序调用优先级高的程序只能通过门重定向，进入门所规定的入口点。这样可以避免低级别的程序代码从任意位置进入优先级高的程序的问题。保护模式下的中断和异常等服务程序也要从“门”进入，80386 的门分为中断门、自陷门和任务门几种。

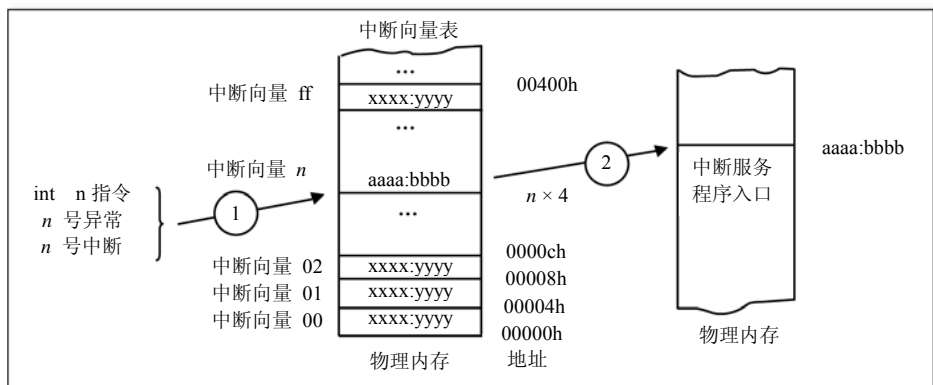


图 1.7 实模式下中断和异常的处理

在保护模式下要表示一个中断或异常服务程序的信息需要用 8 个字节，包括门的种类以及 xxxx:yyyyyyyyy 格式的入口地址等。这组信息叫做“中断描述符”。这样，中断向量表就无法采用和实模式下同样的 4 字节一组的格式。保护模式下把所有的中断描述符放在一起组成“中断描述符表” IDT (Interrupt Descriptor Table)。IDT 不再放在固定的地址 00000h 处，而是采用可编程设置的方式，支持的中断数量也可以设置。为此 80386 处理器引入了一个新的 48 位寄存器 IDTR。IDTR 的高 32 位指定了 IDT 在内存中的基址（线性地址），低 16 位指定了 IDT 的长度，相当于指定了可以支持的中断数量。

如图 1.8 所示，保护模式下发生异常或中断时，处理器先根据 IDTR 寄存器得到中断描述符的地址，然后取出 n 号中断/异常的“门”描述符，再从描述符中得到中断服务程序的地址 xxxx:yyyyyyyyy，经过段地址转换后得到服务程序的 32 位线性地址并转移后执行。

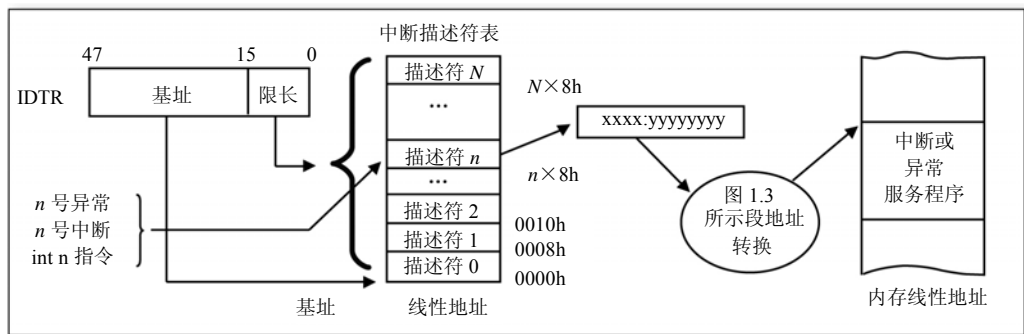


图 1.8 保护模式下的中断和异常处理

由于保护模式下用中断门可以从低优先级的代码调用高优先级的代码，所以不能让用户程序写中断描述符表，否则会引发安全问题（又想到了 CIH 病毒）。这样就如关了窗子

挡住苍蝇，也挡住了微风，用户的系统扩展程序也就不能像在 DOS 中一样再用中断服务程序的方式提供服务了。因为用户程序根本没有权限将中断地址指到自己的代码中来。

在 Windows 中，操作系统使用动态链接库来代替中断服务程序提供系统功能，所以 Win32 汇编中 int 指令也就失去了存在的意义。这就是在 Win32 汇编源代码中看不到 int 指令的原因。其实那些调用 API 的指令就相当于在 DOS 系统中使用 int 指令来完成系统功能。

2. 80386 的保护机制

80286 之前的处理器只支持单任务，操作系统并没有什么安全性可言，计算机的全部资源包括操作系统的内部资源都可以任凭程序员访问。但对于多任务的操作系统，某个捣乱的程序为所欲为会使所有程序都无法运行，所以 80286 及以上的处理器引入了优先级的概念。80386 处理器共设置 4 个优先级（0 级~3 级）。0 级是最高级（特权级）；3 级是最低级（用户级）；1 级和 2 级介于它们之间。特权级代码一般是操作系统的代码，可以访问全部系统资源；其他级别的代码一般是用户程序，可以访问的资源受到限制。

80386 采用保护机制主要为了检查和防止低级别代码的越权操作，如访问不该访问的数据、端口以及调用高优先级的代码等。保护机制主要由下列几方面组成：

- 段的类型检查——段的类型是由段描述符指定的，主要属性有是否可执行，是否可读和是否可写等。而 CS, DS 和 SS 等段选择器是否能装入某种类型的段描述符是有限制的。如不可执行的段不能装入 CS；不可读的段不能装入 DS 与 ES 等数据段寄存器；不可写的段不能装入 SS 等。如果段类型检查通不过，则处理器会产生一般性保护异常或堆栈异常。
- 页的类型检查——除了可以在段级别上指定整个段是否可读写外，在页表中也可以为每个页指定是否可写。对于特权级下的执行代码，所有的页都是可写的。但对于 1, 2 和 3 级的代码，还要根据页表中的 R/W 项决定是否可写，企图对只读的页进行写操作会产生页异常。
- 访问数据时的级别检查——优先级低的代码不能访问优先级高的数据段。80386 的段描述符中有一个 DPL 域（描述符优先级），表示这个段可以被访问的最低优先级。而段选择器中含有 RPL 域（请求优先级），表示当前执行代码的优先级。只有 DPL 在数值上大于或等于 RPL 值的时候，该段才是可以访问的，否则会产生一般性保护异常。
- 控制转移的检查——在处理器中，有很多指令可以实现控制转移，如 jmp, call, ret, int 和 iret 等指令。但优先级低的代码不能随意转移到优先级高的代码中，所以遇到这些指令的时候，处理器要检查转移的目的位置是否合法。
- 指令集的检查——有两类指令可以影响保护机制。第一类是改变 GDT, LDT, IDT 以及控制寄存器等关键寄存器的指令，称为特权指令；第二类是操作 I/O 端口的指令以及 cli 和 sti 等改变中断允许的指令，称为敏感指令。试想一下，如果用户级程序可以用 sti 禁止一切中断（包括时钟中断），那么整个系统就无法正常运行，所

以这些指令的运行要受到限制。特权指令只有在优先级 0 上才能运行，而敏感指令取决于 `eflags` 寄存器中的 `IOPL` 位。只有 `IOPL` 位表示的优先级高于等于当前代码段的优先级时，指令才能执行。

- I/O 操作的保护——I/O 地址也是受保护的對象。因为通过 I/O 操作可以绕过系统对很多硬件进行控制。80386 可以单独为 I/O 空间提供保护，每个任务有个 TSS（任务状态段）来记录任务切换的信息。TSS 中有个 I/O 允许位图，用来表示对应的 I/O 端口是否可以操作。某个 I/O 地址在位图中的对应数据位为 0 则表示可以操作；如果为 1 则还要看 `eflags` 中的 `IOPL` 位，这时只有 `IOPL` 位表示的优先级高于等于当前代码段的优先级，才允许访问该 I/O 端口。

3. Windows 的保护机制

在 Windows 下，操作系统运行于 0 级，应用程序运行于 3 级。因为 Alpha 计算机只支持两个优先级，为了便于将应用程序移植到 Alpha 计算机上，Windows 操作系统不使用 1 级和 2 级这两个优先级。

Windows 操作系统充分利用 80386 的保护机制，所有和操作系统密切相关的资源都是受保护的。运行于优先级 3 上的用户程序有很多限制，只有在写 `VxD`、`WDM` 等驱动程序的时候才可以使用全部资源。在 Win32 汇编编程中要注意避免以下的越权操作（当然写驱动程序不在此列）：

- 显而易见，所有的特权指令都是不可执行的，如 `lgdt`，`lldt`，`lidt` 指令和对 `CRx` 与 `TRx` 等寄存器赋值。但是，读取重要寄存器的指令是可以执行的，如 `sgdt`，`sldt` 和 `sidt` 等。
- Windows 在页表中把代码段和数据段中的内存页赋予不同的属性。代码段是不可写的，数据段中也只有变量部分的页面是可写的。所以虽然可以寻址所有的 4 GB 空间，但访问超出权限规定以外的内存还是会引发保护异常。
- 在 Windows 98 中，系统硬件用的 I/O 端口是受保护的，但其余的则可以操作。如果用户在机器中插了一块自己的卡，用的是 300h 等系统未定义的端口，那么在应用程序中就可以直接操作，但要操作 3f8h（串口）和 1f0h（硬盘端口）等系统已定义的端口就不行了。在 Windows NT 中，任何的端口操作都是不允许的。

如果违反了 Windows 规定的“保护条例”，那么会引发保护异常，处理器会毫不犹豫地把控制权转移到对应的异常处理程序中去。Windows 会在处理程序中用一个很酷的“非法操作”对话框把用户的程序判死刑，没有一点回旋的余地！在 Windows 9x 中，系统有时会用一个蓝屏幕来通知用户程序试图访问不存在的内存页。

如果程序调用的 DLL 中有错，那么错误还是会算在应用程序头上，因为 DLL 的地址空间是被映射到应用程序的空间中去的。Windows 9x 本身是 32 位和 16 位混合的操作系统，为了兼容 DOS 和 Win16 程序，很多的保护措施做起来力不从心。所以系统提供的 DLL 内部反而常常出现越权操作，以至于蓝屏幕不断，这些就不是用户应用程序自己的问题了。

2.1 Win32 可执行文件的开发过程

在 DOS 下，生成一个可执行文件的步骤比较简单，用编译器将源程序编译为 obj 文件，再用链接器将 obj 文件链接成 exe 文件，不同语言的开发过程都差不多。

DOS 可执行文件中的内容是由源程序中所写的代码和数据定义转换而来的。唯一的例外是带覆盖部分（Overlay）的 exe 文件，它在基本的 exe 文件后附加了一些自定义的数据，其中可执行部分的长度由文件头偏移 0002h 和 0004h 中的长度给出，该长度之后到文件实际长度这部分就是 Overlay 部分。这样，即使一个带覆盖的 exe 文件大小远远超过 640 KB，在 DOS 下也能运行，因为操作系统只装入真正的可执行部分，然后由程序自己去读取覆盖部分的数据。一些打包软件生成的奇大无比的自解压包就采用这种结构，可执行部分是解包代码，覆盖部分是被压缩的数据。DOS 对可执行文件覆盖部分的数据格式并没有规定，它是程序员按自己的方式组织的。如果程序员愿意，也可以把这些数据单独放在另外一个文件中。

Win32 可执行文件叫做 PE 文件。PE 文件的基本结构和 DOS 可执行文件有很大的不同。它把程序中的不同部分分成各种节区（Section），其中可以有一个节区是放置各种资源的，如菜单、对话框、位图、光标、图标和声音等（详见第 17 章）。虽然可以把资源部分理解成类似 DOS 可执行文件中的“覆盖”部分，但由于资源是 Win32 可执行文件的标准组成部分，而且是非常重要的组成部分，它的格式是固定的。所以与 DOS 软件的开发过程相比，Win32 软件的开发中多了一个创建资源文件的步骤。

以使用 MASM32 SDK 软件包为例，在用 Win32 汇编开发软件的流程中，程序员要做的工作分创建代码和创建资源两部分，如图 2.1 所示。

代码部分的开发工作与 DOS 下写代码的步骤是一样的。程序员用文本编辑器书写汇编源代码（*.asm 文件）。与 C 源代码类似，asm 文件中也可以用 include 语句包含数据定义和函数声明的头文件，Win32 汇编的头文件一般用 inc 作扩展名。大部分的 include 文件是编译器软件包附带提供的，如 MASM32 SDK 附带的 Windows.inc 文件定义了 Win32 API 中很多参数和数

据结构，其他的 inc 文件则是不同 DLL 中的 Win32 API 函数声明。最后，asm 文件经汇编编译器编译成以 obj 为扩展名的目标文件。

资源文件中可以包括对话框、快捷键、菜单、字符串、版本信息和一些图形资源等内容。资源文件的源文件是一种类似“脚本”的文本文件，它的扩展名一般为 rc，其中用不同的语法定义了不同类型的资源，资源脚本文件最后由资源编译器编译成资源文件*.res。资源脚本文件同样用到很多预定义值，所以软件包中一般也包括资源头文件供源文件来导入。MASM32 SDK 软件包中的资源头文件是 Resource.h。

在资源文件中，不同类型资源的记录方式是不同的。对话框资源只记录定义值，如对话框的大小、位置等，并非真正存储对话框最后显示在屏幕上的像素。这些大小、位置等信息最后由 Windows 解释后才在屏幕上被绘画成像素；菜单、字符串、快捷键等由文本构成；图形资源则真正由像素组成，它们在资源脚本中被定义为一个文件名，由资源编译器从磁盘文件导入。Windows 在资源中支持的图形文件有 bmp 位图文件、cur 光标文件和 ico 图标文件，这些图形文件可以用其他图形处理软件生成。另外，wav 声音文件也可以用在资源中。创建资源的方法在第 5 章中有详细的描述。

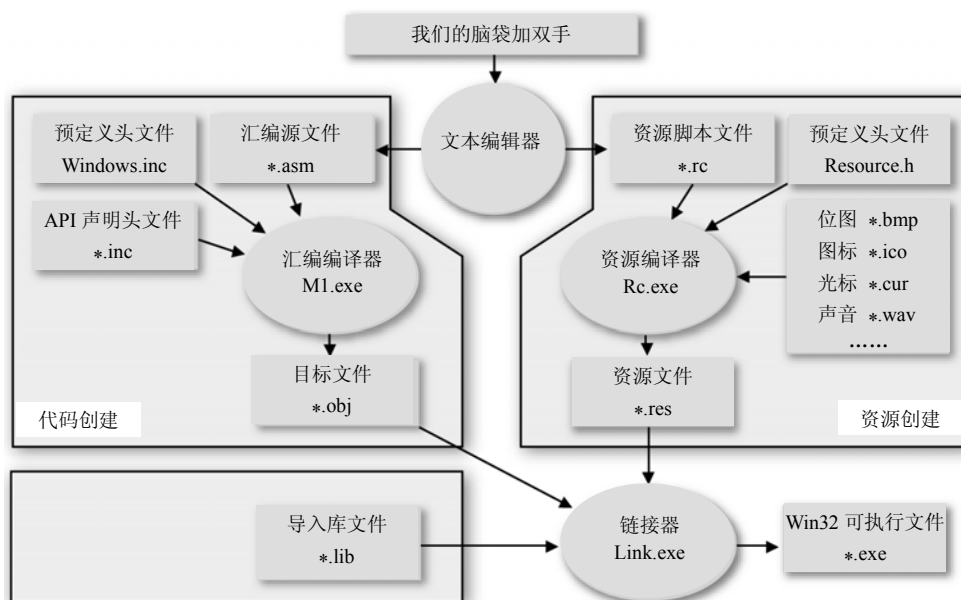


图 2.1 Win32 可执行文件的开发过程

编译好目标文件*.obj 和资源文件*.res 后，最后一步是用链接器将它们链接成可执行文件。链接的时候要用到函数库。在 DOS 环境下编程的时候，使用的函数库是静态库。静态库是一些已经编译好的代码模块。当用户在源程序中用到某个函数的时候，链接器从库文件中将这个函数的二进制代码取出，与 obj 文件合在一起生成最终的 exe 文件。但在 Win32 环境下，大部分的公用函数封装在 DLL 文件中，以动态链接的方式供用户程序调用。这时候库文件中只需要包含函数在 DLL 中的位置信息，不再需要有二进制代码部分。所以链接的时候也只是把库文件中的位置信息取出放入最后的可执行文件中。Win32 中这种只包含位置信息的库文件称为

导入库。动态链接的概念在第 11 章中有详细的描述。

由于 Win32 汇编编程中使用不同汇编编译器的时候，汇编源程序的格式和资源脚本文件的格式可能稍微有所不同。各种头文件、库文件的文件名也有所不同。所以在开始编程之前，必须先选定一种合适的编译器。

2.2 编译器和链接器

选择汇编编译器是开始工作的第一步。不同的编译器用法各不相同，选择合适的编译器可以为开发工作节省很多的时间。这里简单介绍几种不同系列的编译器。常用的汇编编译器有 Microsoft 公司的 MASM 系列和 Borland 公司的 TASM 系列，还有一些小公司推出的或者免费的汇编软件包。

2.2.1 MASM 系列

1. MASM 编译器介绍

MASM 是 Microsoft 公司推出的汇编编译器。它的版本从低到高经过了很多次的升级（微软的通病，升级补丁多如牛毛）。每次升级除了例行的错误修正外都增加了一些新的功能，以至于到最后高版本和低版本的语法和功能相差很多，向下兼容性也不好。低版本的 MASM 固然无法编译高版本的源程序，但高版本的 MASM 也可能无法正常编译低版本的源程序，如 MASM 4.0 写的源程序常常无法在 MASM 6.x 上编译成功。在使用 MASM 系列编译器时，如果不先搞清楚特定的语法和编译选项可以在哪个版本上用，编译中就会错误连篇。所以在这里有必要了解一下 MASM 各版本的演变过程。

表 2.1 列出了不同版本 MASM 编译器的区别。

表 2.1 MASM 编译器各版本的区别

版 本	简 介
MASM 4.00	这是最先广泛使用的一个 MASM 版本，适用于 DOS 下的汇编编程。它很精巧，但使用起来不是很智能化，需要用户自己一板一眼地写出所有的代码。很多教科书上讲的 8086 汇编语法都是针对这个版本的，对程序员来说，它只比用 Debug 方便一点点
MASM 5.00	MASM 5.00 比 4.00 在速度上快了很多，并将段定义的伪指令简化为类似 .code 与 .data 之类的定义方式，同时增加了对 80386 处理器指令的支持，对 4.00 版本的兼容性很好
MASM 5.10	对程序员来说，这个版本最大的进步是增加了对 @@ 标号的支持。这样，程序员可以不再为标号的起名花掉很多时间。另外，MASM 5.10 增加了对 OS/2 1.x 的支持
MASM 5.10B	1989 年推出，比上一个版本更稳定、更快，它是传统的 DOS 汇编编译器中最完善的版本
MASM 6.00	1992 年发布，有了很多的改进。编译器可以使用扩展内存，这样可以编译更大的文件，可执行文件名相应从 Masm.exe 改为 ML.exe。从这个版本开始可以在命令行上用 *.asm 同时编译多个源文件，源程序中数据结构的使用和命令行参数的语法也更像 C 的风格。最大的改进之一是开始支持 .if/.endif 这样的高级语法，这样，使用复杂的条件分支时和用高级语言书写一样简单，可以做到几千行的代码中不定义一个标号；另外增加了 invoke 伪指令来简化带参数的子程序调用。这两个改

	进使汇编代码的风格越来越像 C，可读性和可维护性提高了很多
续表	
版 本	简 介
MASM 6.00A	未发售的版本
MASM 6.00B	最后一个支持 OS/2 的 MASM 版本，修正了上一版本中的一些错误
MASM 6.10	修正了一些错误，同时增加了 /Sc 选项，可以在产生的 list 文件中列出每条指令使用的时钟周期数
MASM 6.10A	1992 年发布，修正了一些内存管理方面的问题
MASM 6.11	1993 年 11 月发布，支持 Windows NT，可以编写 Win32 程序，同时支持 Pentium 指令，但不支持 MMX 指令集
MASM 6.11C	1994 年发布，增加了对 Windows 95 VxD 的支持
MASM 6.12	1997 年 8 月发布，增加了 .686，.686P，.MMX 声明和对相应指令的支持
MASM 6.13	1997 年 12 月发布，增加了 .K3D 声明，开始支持 AMD 处理器的 3D 指令
MASM 6.14	这是一个很完善的版本，它在 .XMM 中增加了对 Pentium III 的 SIMD 指令集的支持，相应地增加了 OWORD（16 字节）的变量类型
MASM 6.15	2000 年 4 月发布

不同版本 MASM 产生的 obj 文件的格式也不相同，在 DOS 和 Win16 时期，Microsoft 使用的 obj 文件格式为 OMF 格式（Intel Object Module Format），到了 Win32 时期后改用了 COFF 格式（Common Object File Format），原因之一是 COFF 格式更像最终的 PE 文件，在链接的时候可以去做更少的处理，MASM 从 6.11 版本开始支持 COFF 格式。

用 Microsoft 的产品编写 Win32 程序，不管是使用 VC 还是 MASM，都必须使用 COFF 格式，因为 Microsoft 的 32 位的 Link 只支持将 COFF 格式的 obj 文件链接成 PE 文件，另外所有的导入库等支持文件的格式也全部是 COFF 格式的。

单独的 MASM 软件包不是免费的，但免费发布的 Windows 98 DDK 中却包括完整的 MASM 6.11d 版本，Win98ddk.exe 文件一开始可以在 Microsoft 的网站上免费下载，现在已经不再提供，但读者还可以在一些第三方的站点找到这个软件包。

注意：整个 Win98ddk.exe 文件有 18 MB 之大！得到了 MASM 6.11d 之后，可以从 Microsoft 获取升级软件一直升级到最新的版本，升级包的下载地址是：

ftp://ftp.microsoft.com/softlib/mslfiles

升级包的文件名和版本号相对应，如到 6.14 版本的升级文件是 Ml614.exe，迄今为止最新的 MASM 6.15 版本可以从 Visual C++ 6.0 Processor Pack 中获取，该软件的下载地址可以在微软的网站上查找。

2. Ml.exe 的用法

不同版本的 MASM 在使用上有很大的不同，本节所指的是可用于 Win32 汇编编程的 MASM 6.14 及以上版本，MASM 编译器的命令行用法为：

Ml [/选项] 汇编源文件列表 [/link 链接选项]

要注意的是汇编选项要集中写在源文件名的前面，比如下面的两条命令：


```
Ml /c /coff /Cp Test.asm
Ml /c /coff Test.asm /Cp
```

虽然它们都可以编译 Test.asm 文件,但第二句的/Cp 选项由于写在了汇编源文件名的后面,实际上会被忽略掉。另外,多个选项之间一定要加空格,经常有初学者将多个选项连在一起写成“/c/coff”,结果当然是会报错,因为编译器将它当做一个选项来辨认了。

Ml 在 Win32 汇编编程中常用的选项如表 2.2 所示。

表 2.2 Ml 的常用选项

选 项	简 介
/c (常用)	仅进行编译,不自动进行链接
/coff (必用)	产生的 obj 文件格式为 COFF 格式
/Cp (常用)	源代码区分大小写
/Fo filename	指定输出的 obj 文件名
/Fe filename	指定链接后输出的 exe 文件名
/Fl [filename]	产生 .lst 列表文件
/Gc	函数调用类型用 FORTRAN 或 PASCAL 形式
/Gd	函数调用类型用 C 语言形式
/Gz (常用)	函数调用类型用 StdCall 形式
/I pathname	指定 include 文件的路径
/link 选项	指定链接时使用的选项
/Sc	在列表文件中列出指令的时钟周期
/Zi	增加符号调试信息

与用 MASM 5.0 及以下的版本编写 DOS 程序相比,用 MASM 的高版本编写 Win32 程序有几个必须使用的选项,如/coff 等。另外,用/Zi 增加调试信息在源码级调试中也很有用。

3. Link 的用法

用 Ml.exe 编译的 COFF 格式的 obj 文件可以用 Link.exe 链接成可执行 PE 文件,Microsoft 的 Link.exe 有两个系列的版本,用于链接 DOS 程序的链接器为 Segmented Executable Linker;可以链接 Win32 PE 文件的链接器为 Incremental Linker,这里指的是 Incremental Linker 的用法。

Link 的命令行使用方法为:

```
Link [选项] [文件列表]
```

命令行参数中的文件列表用来列出所有需要链接到可执行文件中的模块,可以指定多个 obj 文件、res 资源文件以及导入库文件。Link 的选项很多,常用的选项如表 2.3 所示。

表 2.3 Link 的常用选项

选 项	简 介
/BASE: 地址	指定程序装入内存的基地址,一般 PE 文件默认的装入地址是 0x400000 处,dll 文件装入 0x10000000,用此选项可以修改这个默认值

续表

选 项	简 介
/COMMENT: 注释	在 PE 文件的文件头后面加上文本注释, 想在可执行文件中加入版权字符串可以用这个办法, 如果字符串中包括空格, 那么要在头尾加双引号
/DEBUG	在 PE 文件中加入调试信息
/DEBUGTYPE: 类型	加入的调试信息类型, 可以是 CV 或 COFF
/DRIVER: 类型	链接 Windows NT 的 WDM 驱动程序时用, 类型可以是 WDM 或者 UPONLY
/DLL	生成动态链接库文件时用
/DEF: 文件名	编写链接库文件时使用的 def 文件名, 用来指定要导出的函数列表
/ENTRY: 标号	指定入口标号
/IMPLIB: 文件名	当链接有导出函数的文件时 (如 DLL) 要建立的导入库名
/INCREMENTAL:ON OFF	是否增量链接, 增量链接只重写可执行文件自上次链接后改动的部分, 所以可以增加链接速度, 但会增加文件长度
/LIBPATH: 路径	指定库文件的目录
/MACHINE:平台名称	指定输出的可执行程序运行平台, 可以是 ALPHA, ARM, IX86, MIPS, MIPS16, MIPS16R41XX, PPC, SH3 和 SH4 等
/MAP: 文件名	生成 MAP 文件
/OUT: 文件名	指定输出文件名, 默认的扩展名是 .exe, 如果要生成其他文件名, 如屏幕保护*.scr 等, 则在这里指定一个具体的文件名
/RELEASE	填写文件头中的校验字段
/SECTION: 节区, 属性	改变节区的属性, 如 exe 文件中代码节区的属性通常是不可写的, 用户也可以在这里将它设置为可写, 属性可以是: E, R, W, S, D, K, L, P 和 X 等
/STACK: 尺寸	设定堆栈尺寸
/STUB: 文件名	这是一个有趣的参数, Win32 文件有个简单的 DOS 文件头, 以便在 DOS 下执行时打出“必须在 Windows 下执行”一类的消息, 这部分称为 DOS STUB, 用户可以在这里指定用一个 DOS 可执行文件代替它, 例如, 用 DOS 的 FDISK.EXE 代替, 那么程序在 Windows 下运行的会是用用户编写的代码, 但在 DOS 下运行的就是 FDISK.EXE
/SUBSYSTEM: 系统名	指定程序运行的操作系统, 可以是 NATIVE, WINDOWS, CONSOLE, WINDOWSCE 和 POSIX 等
/VXD	编写 Windows 95 VxD 驱动程序时指定

由表 2.3 可见, Link 的选项远比 MASM 要复杂, 但并不是所有的选项都是频繁使用的, 编写普通的 Win32 可执行文件时, 必须用的选项只有 /subsystem 一个, 其他的都可以用默认值。

一般来说, 用 MASM 编译和链接一个 Win32 汇编源程序常用的命令是:

```

Ml /c /coff xx.asm
Link /subsystem:windows xx.obj yy.lib zz.res    (普通 PE 文件)
Link /subsystem:console xx.obj yy.lib zz.res   (控制台文件)
Link /subsystem:windows /dll /def:aa.def xx.obj yy.lib zz.res (DLL 文件)

```

在 Ml 中使用 /c 选项表示只生成 obj 文件而不是直接产生 exe 文件, 原因是链接的时候可能需要指定资源文件, 所以不能让 Ml 直接用默认的方式链接; /coff 选项是必需的, 因为链接器只支持 COFF 格式的 obj 文件, 其他的选项, 如 /Cp 和 /Gz 虽然也是必需的, 但是由于可以在 asm 源文件中用伪定义设置, 所以一般不在命令行中指定, 以免遗漏。

使用 Link 的时候, /subsystem 选项必须被指定, 一般指定为 windows, 当编写控制台程序的时候要改为 console。写 dll 的时候要用 /def 指定列表定义文件, 同时要指定 /dll 选项。其他的一些参数如 /stub, /section 和 /base 等只在编写特殊用途的程序时才使用。

2.2.2 TASM 系列

1. TASM 的用法

TASM 是 Borland 公司推出的汇编编译器, 也是一种使用很广泛的编译器, 与 MASM 相比, TASM 的升级没有这么频繁。TASM 早在 1.0 版本就有了对 80386 处理器指令的完全支持 (MASM 要到 5.0 版本才支持 80386 指令), 1989 年推出的 1.01 版本修正了 1.0 版的一些错误; 早期的版本还有 TASM 3.0 和 TASM 4.0, 其中 4.0 版是 TASM 系列编译器编写 DOS 程序使用最广泛的版本。

到目前为止, TASM 的最后一个版本是 5.0 版, 这个版本支持 Win32 编程, 并单独为 Win32 编程附带有一整套的 32 位程序: 32 位的编译器 TASM32.EXE、链接器 TLINK32.EXE 和资源编译器 BRC32.EXE。与这些 32 位程序对应的 16 位工具在软件包中依然存在, 文件名为 TASM.EXE, TLINK.EXE 和 BRC.EXE 等。

TASM 5.0 命令行的使用方法是:

TASM32 [选项] 源文件名[, [目标文件名], [列表文件名], [索引文件名]] [;]

在 Win32 编程时 TASM 的常用选项如表 2.4 所示。

表 2.4 TASM 5.0 的常用选项

选 项	简 介
/ml, /mx, /mu	对大小写是否敏感: ml=全部敏感, mx=全局变量大小写敏感, mu=不敏感, Win32 编程中必须用 ml 选项
/m#	允许第#次编译扫描时可以向前引用, 一般使用/m2
/i 路径	设定 include 文字所在的路径
/l, /la	Lst 文件的格式: /l 为正常格式, /la 为扩充格式
/zi, /zd, /zn	符号调试信息的控制: /zi 为全部, /zd 为仅有行号, /zn 为不产生调试信息

TASM 和 MASM 之间的导入库和目标文件等不能通用, 程序员无法用 Microsoft 的链接器来链接 TASM 产生的 obj 文件, 反之亦然, 这是因为 TASM 的库文件和 obj 文件一直是 OMF 格式, 与 Microsoft 使用的 COFF 格式不兼容。

TASM 不是免费软件, 无法从 Borland 的网站上直接下载, 但在网上随处可以找到 TASM 5.0 版本的下载链接, Borland 的官方网站中仅提供了 5.0 版本到 5.0r 版本的升级包。

2. TLINK 的用法

与 TASM32.exe 配合的链接器为 TLINK32.exe, 它的用法是:

TLINK32 选项 目标文件列表, 输出文件, map 文件, 库文件, def 文件, 资源文件

指定这些文件名的时候, 中间必须用逗号隔开, 如果不想指定某个文件名, 可以在逗号中

间留空，TLINK32 的常用选项如表 2.5 所示。注意，TASM32 的选项是用斜杠开头，TLINK32 的选项却用减号开头。

表 2.5 TLINK32 的选项

选 项	简 介
-c	链接时区分大小写
-B:xxxx	指定可执行文件装入内存的基地址
-Txx	输出文件类型，-Tpe 表示输出 PE 类型的 exe 文件，-Tpd 表示输出 PE 类型的 dll 文件
-ax	文件类型，-aa 表示使用 Windows API，-ap 表示使用兼容代码
-v	在输出文件中包括调试信息

下面是用 TASM 编译和链接一个 Win32 汇编源程序的常用命令：

```
TASM32 /ml /m2 xx.asm
TLINK32 -Tpe -aa -c xx.obj,,,yy.lib,,zz.res      (普通 PE 文件)
TLINK32 -Tpd -aa -c xx.obj,,,yy.lib,aa.def,zz.res (DLL 文件)
```

由于 Windows API 区分大小写，所以 TASM32 的/ml 和 TLINK32 的-c 选项必须指定，TLINK32 中的-Tpe 和-aa 选项也必须指定，否则链接出来的就不是 Win32 可执行文件了。其他的选项如调试信息等则可以根据需要选择使用。

2.2.3 其他编译器

除了 MASM 和 TASM 这两种主流的汇编编译器，汇编编程中还可以用到一些其他的编译器，这些编译器大部分是免费的，如表 2.6 所示。

表 2.6 常用汇编编译器列表

编 译 器	简 介	支持编程
NASM	Netwide/National Assembler，开放源代码的免费软件，使用传统的 Intel 语法	DOS，Win32，Linux
FASM	支持 8086-80486/Pentium/MMX/SSE/SSE2 指令，16/32 位代码	DOS，Win32
SpAsm	Specific Assembler，用于 ReactOS/Win32 的汇编编译器，有简化的语法和宏指令，自带 IDE 环境	Win32
VisualASM	附带 IDE 环境	Win32
Pass32	支持面很广的编译器，一个显著的特点是支持 DOS extender，可以直接将 DOS extender 链接到可执行文件中	DOS，Win32，DOS DPMI
GASM	GNU Assembler，兼容 NASM，可以用于 DOS 下的保护模式编程	DOS
Nbasm	NewBASIC++ Assembler，适合于建立小规模汇编程序，用来学习汇编是很不错的	DOS
CHASM	Cheap Assembler，用于 MS-DOS 编程的共享软件	DOS

值得一提的是 NASM，这个编译器也支持 Win32 汇编，不同于 MASM 和 TASM 这两个编译器，它是免费软件并且开放源代码，如果读者对编译器的原理感兴趣的话，可以从网上下载整个 NASM 的软件包来看一看，NASM 的官方站点网址是 <http://www.nasm.us>。

NASM 不具有 MASM 和 TASM 所拥有的一些高级语法，如将带参数的调用语句自动转化

成多个 `push` 指令和一个 `call` 指令，更没有 MASM 所有的 `.if/endif` 等高级语法，这使 NASM 用于 Win32 编程相当不方便，整个感觉和用 MASM 4.0 差不多，几乎所有的细节都需要用户自己写。但 NASM 的一个显著优点部分地抵消了这个缺点，因为它支持不同的平台，如 Windows, Linux 和 OS/2 等，用它写 Win32 程序虽然有些麻烦，但熟悉了它的语法后可以很快在 Linux 的汇编中上手，所以使用 NASM 的程序员还是不少。

2.2.4 MASM , TASM 还是 NASM

既然编写 Win32 汇编可以使用的编译器有这么多种，那么我们究竟使用哪一种呢？单从编译器的角度来说，用 MASM 写汇编程序是最方便的，支持 `@@` 标号，用 `invoke` 调用子程序，支持局部变量和有 `.if/else /endif` 高级语法等优点就已经是足够的理由了，更不用说有 Microsoft 这个强大的后盾了，这是其他的编译器所无法比拟的，但使用 MASM 的不方便之处是它从来就不是当做一个完整的软件包发售的，要开始用 MASM 写 Win32 程序还要费很大的周折。

首先是不同版本的 MASM 软件包中都没有包含资源编译器，资源编译器是当做 Windows SDK 的一部分发行的，或者要从 Microsoft Visual Studio 软件包的 Common 目录中找，更有甚者，和 MASM 软件包一同发售的 Link 程序竟然不是 32 位的，只能用来链接 DOS 程序，即使是 6.11 以上版本中也是如此。迄今为止，MASM 软件包中附带的链接程序全是 Segmented Executable Linker, Incremental Linker 只能在 Microsoft Visual Studio 软件包的 Visual C++ 目录中找到。

其他一些有用的工具也没有包含在软件包中，如库管理工具和 `make` 工具等，所以要使用 MASM 进行 Win32 汇编编程就要对软件包进行改造：一方面舍弃 MASM 软件包中附带的 Link 程序；另一方面，需要到其他地方去找资源编译器和 32 位的链接器等工具软件。

对初学者来说，连 Win32 汇编的开发要哪几个步骤及需要什么软件都还没有底，就更不用说从不同的工具包中寻找需要的软件了（而且必须是合适的 32 位版本）；另外，Win32 编程用到的导入库在 MASM 软件包中并没有包括，同样要到 Visual C++ 中去找；最大的障碍在于：MASM 软件包中没有头文件，也不可能直接使用 Visual C++ 的头文件，所有这些头文件必须自己根据资料以及参考 Visual C++ 的 `.h` 文件整理出来，而 Windows 的数据结构和预定义的数据是出了名的多，所有这些让使用 MASM 编写 Win32 汇编程序非常难以下手。

从工具包的完整性来说，TASM 和 NASM 相对来说要好一点，TASM 软件包中包括了 32 位的资源编译器和链接器，也有一个 32 位的导入库文件，这样，用户不用添加任何其他软件就可以直接用 TASM 写出完整的 Win32 程序。但 TASM 软件包中也没有 Windows 数据结构和预定义的头文件，所有资料同样需要用户自己整理。

另一方面，与 MASM 相比，TASM 在优化方面做得不是很好，简单举几个例子：比如 TASM 无法处理大量的预定义，如果用户把所有的预定义整理到 `Windows.inc` 文件中，然后在源文件中包括进来，编译的时候就会出现 “Out of hash space” 错误，结果每次只好把要用到的定义分拣出来写成一个小的 `include` 文件；再比如在源程序中用 `extrn` 定义 API 函数，不管在程序中实际有没有用到这个函数，TASM 都会在最后 `.exe` 文件的导入表中加上这个函数名，这就意味着无法用

偷懒的办法把所有的 API 函数声明写到一个 include 文件中，除非用户可以忍受可执行文件中无效的字节数比有效的多得多！还有一个缺点是：TASM 在定义结构的时候，不同的结构中不能有同名的字段，而 Windows 的数据结构定义出奇的多，结果不同结构中的同名字段要在前面加上一些前缀以示区别，这就会使源代码中的结构定义和参考资料中的结构定义在字面上不符合，使用时还要不停地去看结构中的字段究竟是怎么定义的。这些小缺点使 TASM 在使用时程序员做的无效工作比有效工作还多。

NASM 的优点就是学到的语法可以直接用在 Linux 的汇编中，缺点也是显而易见的，就是免费软件往往缺乏强大的后盾，开发的力度肯定不如大的软件公司，具体就表现在 NASM 中几乎没有一点可以帮程序员省心的高级语法，而这些恰恰是编写高可维护性程序所必须具有的特征。而且，用 NASM 编程同样存在用户自己整理数据结构定义和预定义的问题。

比较这些编译器，可以发现很难找到直接拿来就可以开始写 Win32 汇编程序的软件包，因为每个软件包中都没有关键的头文件，而用户自己整理头文件不仅使程序的可移植性大打折扣，而且工程量之大使程序员只能写很小规模的程序，所以，理想的软件包应该是这样的：

- 包含所有所需工具，如汇编编译器、资源编译器和链接器等。
- 编译器支持高级语法，使源程序便于维护。
- 包含完整的头文件，如 Windows 的数据结构定义和预定义等。
- 包含齐全的导入库。
- 有大量的例子和说明文档。

有这样的汇编软件包吗？有！MASM32 SDK 软件包就是我们的选择。

2.2.5 我们的选择——MASM32 SDK 软件包

读者可能会感到奇怪，怎么又出来一个 MASM32 SDK，这是什么公司的产品呢？实际上，MASM32 SDK 是不同工具软件的大集合，它的汇编编译器用的是微软 MASM 软件包中的 Ml.exe，资源编译器和 32 位链接器使用的是 Microsoft Visual Studio 中的 Rc.exe 和 Link.exe，同时包含了 Microsoft Visual Studio 中的其他一些工具，如 Lib.exe 和 DumpPe.exe 等，所有的工具都是适合于 Win32 编程的版本。

同时，MASM32 SDK 软件包包括了详尽的头文件和导入库文件，导入库文件取自 Visual C++ 的导入库，规模庞大的头文件则是发布者整理的，软件包中还包括了很多的例子，涉及 Win32 汇编的很多方面，例子收集自世界各地 Win32 汇编爱好者发布的源程序。为了使工具包更实用，发布者还为它编写了一个简单的 IDE 环境，包括一个专用的汇编源程序编辑器和源程序模板生成器等。

MASM32 SDK 软件包使汇编不再只用来编写简单的程序和少量的核心模块，它的目标完全是为了用汇编写出专业的大型程序。虽然它是一个大杂烩，但发布者做了所有汇编程序员都想做、却又在庞大的工程量前止步的工作——收集合适的工具软件、收集导入库、整理出完整的头文件、收集例子文件、写帮助文档……

让我们感谢发布者 Steve Hutchesson 为所有的 Win32 汇编程序员所做的这一切。

迄今为止, MASM32 SDK 的最高版本是 MASM32 SDK Version 10 (简称 MASM32V10), 与早一些的 MASM32V7 和 MASM32V6 版本相比, 使用的编译器等可执行文件并没有什么改变, 不同的地方是在头文件中增补了一些数据结构定义和增加了不少例子程序。最新版本的 MASM32 SDK 软件包可以在官方网站 <http://www.masm32.com> 中下载。MASM32 SDK 是一个免费的软件包, 但其中的不同部分如编译器和例子程序等可能属于不同的公司和个人, 使用时需要遵从他们的版权声明。

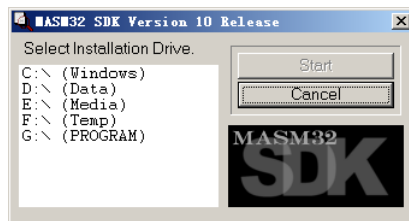


图 2.2 MASM32 的安装界面

MASM32 SDK 的安装界面如图 2.2 所示。

安装 MASM32 SDK 时, 在选择目标驱动器后, 工具包会被安装到根目录下的 MASM32 目录中。读者需要知道一些重要文件的位置, 这在使用时非常重要, MASM32 下各目录的列表和说明如表 2.7 所示。

表 2.7 MASM32 SDK 软件包的安装目录说明

目 录	介 绍
\masm32	IDE 环境、内带的文本编辑程序和模板生成程序等
\masm32\include	所有的头文件, Windows.inc 为数据结构和预定义值的定义文件, Resource.h 为资源文件的头文件, 其他 .inc 文件为对应同名 DLL 文件中的 API 函数声明文件
\masm32\lib	所有的导入库文件, 每个 .lib 文件是对应 DLL 文件的导入库
\masm32\bin	可执行文件目录, 里面包括 Ml.exe, Link.exe 和 Rc.exe 等
\masm32\help	帮助文件目录
\masm32\m32lib	一些常用 C 子程序的汇编实现源程序, 如熟悉的 stdin 和 stdout 等, 有一定的参考价值
其他目录	主要为例子和可用可不用的小工具, 例子广泛收集自网上不同作者的作品, 很有参考价值

如果不用内带的 IDE 环境, 不看附带的例子和帮助文件, 那么有了 bin, include 和 lib 这三个目录中的内容, 读者就可以进行 Win32 汇编编程了, 其他目录中的文件仅起辅助作用。

本书的编程环境就是以 MASM32 SDK 软件包为基础的, 事实上, 现在 MASM32 SDK 已经是最流行的 Win32 汇编开发包, 世界上大部分的 Win32 汇编程序员都用它来进行 Win32 软件开发。

2.3 创建资源

2.3.1 资源编译器的使用

资源编译器用来把资源脚本文件 (*.rc) 编译成资源文件 (*.res), MASM32 SDK 软件包中使用的是 Visual C++ 附带的 Rc.exe 程序, TASM 软件包中使用的是 BRC32.exe 或 BRCC32.exe, 两者的用法都比较简单, 它们有相同的命令行语法:

Rc 或 BRC32 [选项] 资源脚本文件名

Rc 和 BRC32 在使用中没有必需的选项，不像汇编编译器一样必须使用一些关键的选项。如果编译成功，就会产生以 res 为扩展名的资源文件，两者生成的资源文件的格式是一样的。

资源文件编写是 PE 开发的标准步骤，由于不同的语言使用的资源编译器，以及生成的 .res 文件格式都是一样的，没有汇编格式的资源文件和 C 格式的资源文件之分，所以汇编开发包中的资源编译器实际上就是 C 开发包中的资源编译器。由于 C 语言的使用远比汇编广泛，所以资源脚本文件的语法是 C 格式的，如等值定义语句使用 #define 而不是汇编常用的 equ，注释使用 “//” 而不是 “;”，头文件习惯使用 .h 扩展名而不是 .inc，参数定义有 “或” 操作时使用 “|” 操作符而不是汇编的 “or” 操作符等，这些在使用中必须注意，否则会引起语法错误。

两种资源编译器在使用中稍微有所区别，由于 BRC32.exe 内部可以解释 Windows 的一些预定义值，所以不用附带头文件，只有遇到最新的预定义时才需要头文件，而 Rc.exe 并没有这个功能，所以在脚本文件中必须把头文件 Resource.h 包括进去。

2.3.2 所见即所得的资源编辑器

资源脚本文件中一个典型的对话框定义是这样的：

```
DLG_MAIN      DIALOG 0, 0, 176, 66
STYLE         DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION      "对话框模板"
FONT         9, "宋体"
{
    DEFPUSHBUTTON "退出", IDOK, 120, 46, 50, 14
    CONTROL "", -1, "Static", SS_ETCHEDHORZ | WS_CHILD | WS_VISIBLE, 7, 38, 164, 2
}

```

第一句定义了对话框的左上角坐标为 (0,0)，大小为 (176,66)，中间的 DEFPUSHBUTTON 定义了一个位置为 (120,46)，大小为 (50,14) 的按钮，看到这些定义后脑袋里要浮现一个正确的对话框是很不容易的，只能在纸上打格子画出来后才能明白它的模样，所以直接用文本编辑器书写资源脚本文件有诸多不便，就像谁都不会用 db 语句一个个像素地定义 bmp 位图一样，写资源脚本同样需要像图形编辑软件一样所见即所得的工具。Borland 公司的 Resource Workshop 和 VC 环境内带的资源编辑器就是这样的工具。

1. Resource Workshop 资源编辑器

Resource Workshop 是 Borland 公司出品的资源编辑器，它是当做 Borland C++ 的一个组成部分发布的，并不是一个单独的产品，但由于它使用方便，应用一直比较广泛，于是有人将它单独分离出来做成一个软件包下载，Resource Workshop 的初始版本不是双字节版，所以无法支持中文，在编辑的时候输入中文，存盘后会变成乱码，非常不便，还是有“好事者”将它修改成了双字节版供人下载，在网上搜寻下载的时候要注意版本区别。

同样是上面举例的对话框，在 Resource Workshop 中的编辑界面如图 2.3 所示，是不是形象多了？Resource Workshop 不但支持资源脚本文件 *.rc，同时支持资源文件 *.res，读者可以用它打开 *.res 文件，编辑后存为 *.rc 文件，同时也可以直接用它编辑 *.rc 文件且生成 *.res 文件，

这样就可以免去用资源编译器编译脚本文件的步骤。

奇怪的是, BRCC32.exe 不需要预定义头文件的支持, 而同是 Borland 公司的 Resource Workshop 却没有内置窗口风格等定义值, 所以在使用中需要 Resource.h 头文件的支持, 并且, Resource.h 文件必须和 .rc 文件在同一个目录中, 所以用它来编辑资源脚本文件的时候要注意拷贝一份 Resource.h 文件, 当然, 如果直接用它来编辑资源文件*.res 则不需要头文件。

Resource Workshop 的缺点是版本比较老, 毕竟它是随早期的 Borland C++发布的, 所以它不支持一些新的特征, 如窗口的扩展风格等, 如果在 .rc 文件的定义中用到这些风格, Resource Workshop 会提示不认识这些关键字, 用它打开含扩展风格对话框的 .res 文件, 则会提示一个 Unexpected file format 错误。

2. 用 Visual C++编辑资源

Visual C++本身是一个很大的软件包, 里面集成了资源编辑功能, 它也是所见即所得的编辑工具, 并且支持最新的资源特征, 如最新的对话框风格和一些新的控件等, 同时它是双字节版本, 不必担心乱码问题, Visual C++也支持编辑脚本文件*.rc 和资源文件*.res, 可以在两者之间互相转换。单从功能方面考虑, 用 Visual C++来编辑资源是一个很好的选择。Visual C++的资源编辑界面如图 2.4 所示。

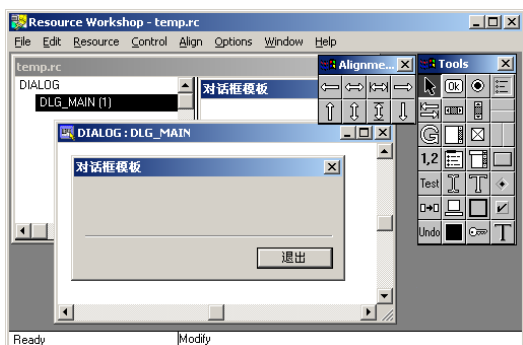


图 2.3 Resource Workshop 的使用界面

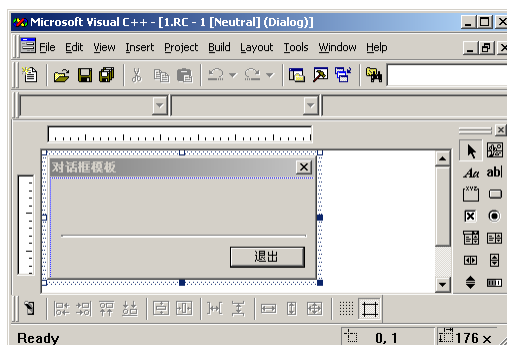


图 2.4 Visual C++的资源编辑界面

使用 Visual C++做资源编辑器的一个显著缺点就是它的规模, 资源编辑的功能是集成在 IDE 环境中的, 要使用它就要安装整个 Visual C++软件包, 至少需要几百 MB 的空间!而 Resource Workshop 只有几 MB, MASM32 SDK 软件包也只有不到 10 MB, 为了一个资源编辑功能用去几百 MB 的空间似乎有点好笑。

另外, 用 Visual C++生成的.rc 文件总是包含了很多 VC 自己的头文件, 如果将它们去掉, 下次就无法再用 VC 打开; 如果不去掉这些多余的信息, 那么用 Rc.exe 编译的时候就要把所有需要的头文件拷贝过来, 将源程序和别人交流的时候, 别人要编译这个资源脚本也必须到 VC 中去找这些头文件。

但 Visual C++毕竟是个功能强大的工具, 建议读者还是使用 Visual C++来编辑资源, 存盘的时候直接存为*.res 文件, 这样可以省去编译资源的步骤。到最后调试完成的时候或者需要交流的时候, 可以保存一份 .rc 文件并将文件中 VC 使用的多余内容去掉, 整理成 Rc.exe 可以编

译的格式。

2.4 make 工具的用法

2.4.1 make 工具是什么

在 DOS 时期编写汇编程序的时候，编译器和链接器基本上不用什么参数，命令只有区区两条：

```
Masm xxx.asm;  
Link xxx.obj;
```

只要做个批处理把 xxx 换成 %1，然后在命令行键入 asm.bat xxx 就万事大吉了，很是方便。Win32 编程就不一样了，不管编译器还是链接器都需要加上必要的选项，文件列表也多了起来，如链接器的命令行参数中要列出 obj，lib，res 和 def 等多种文件，又多了资源编译这一步，如果用批处理实现，要加的参数太多太乱，而每次用手工一行行地键入命令的话，那对程序员来说简直就是一场灾难。当然，一种简单的解决办法就是为每个编程项目单独建立一个批处理，每次改动后，运行批处理把所有模块重新编译一次，但是当程序很庞大的时候，这将花费很长时间，那么该如何处理呢？这时候就要用到 make 工具来维护代码了，从网上下载 Win32 汇编的例子程序时，常常发现除了 *.asm 和 *.rc 文件外，例子文件包中往往还有一个 makefile 文件，这就是给 make 工具用的。

make 工具可以看成是一个智能的批处理工具，它本身并没有编译和链接的功能，而是用类似于批处理的方式——通过调用 makefile 文件中用户指定的命令来进行编译和链接的。但是，批处理会执行全部命令将全部源文件编译，包括那些不必重新编译的源文件，而 make 工具则可根据目标文件上一次编译的时间和所依赖的源文件的更新时间自动判断应当编译哪些源文件，对没有更新过的文件不会处理，这样就可以大大提高程序调试的效率。

举例说明，我们要写一个 test.exe 文件，生成最后的可执行文件有 4 个步骤：

- (1) 汇编源文件 x.asm，其中用到头文件 common.inc，它们经 Ml.exe 编译成 x.obj；
- (2) 汇编源文件 y.asm，用到头文件 common.inc 和 y.inc，它们经 Ml.exe 编译成 y.obj；
- (3) 资源脚本文件 x.rc，经 Rc.exe 编译成 x.res；
- (4) 最后用 Link 将 x.obj，y.obj 和 x.res 链接成 test.exe。

可以看出，当程序调试的时候，如果修改了 x.asm，也就是说 x.obj 的文件时间比 x.asm 要早，就需要重新执行步骤 (1) 和 (4)；如果修改了 y.asm 或 y.inc，那么需要重新执行步骤 (2) 和 (4)；如果修改的是 x.rc，则步骤 (3) 和 (4) 必须重新执行；如果修改的是 common.inc，因为 x.asm 和 y.asm 都和它有关，所以步骤 (1)、(2) 和 (4) 都要重新执行；如果同时修改了 common.inc 和 x.rc，那么必须重复全部步骤。在这个例子中，文件的依赖关系就是：

- test.exe 依赖于 x.obj，y.obj 和 x.res；

- x.res 依赖于 x.rc;
- x.obj 依赖于 x.asm 和 common.inc;
- y.obj 依赖于 y.asm, common.inc 和 y.inc。

make 可以根据文件的时间正确判断文件的新旧并执行相应的步骤。但 make 又是如何知道文件之间的依赖关系呢？这需要用户用一个描述文件来指定。前面提到的 makefile 就是这个描述文件，执行 make 工具的时候，它会默认用 makefile 做描述文件名来进行相应的工作，书写描述文件有规定的语法，虽然语法不是很简单，但写好以后就省事多了。

Microsoft 的 make 工具文件名为 nmake.exe，它并不是 MASM 软件包的一部分，但可以在 Visual C++ 的 Bin 目录下找到。Borland 公司的 make 工具文件名是 make.exe，它已经包括在 TASM 5.0 工具包中。两者默认的描述文件名都是 makefile，描述文件的语法也大同小异，只是使用时命令行参数有些不同。

2.4.2 nmake 的用法

在命令行键入 nmake /? 可以显示帮助信息，nmake 的语法如下：

```
nmake [选项] [/f 描述文件名] [/x 输出信息文件名] [宏定义] [目标]
```

说明如下：

- /f 参数——如果描述文件名不使用默认的 makefile，可以用/f 参数指定。
- /x 参数——如果想把屏幕输出的信息存到一个文件中，可以用/x 参数指定（用 DOS 下的管道操作符 nmake > 文件名的方法无效）。
- 宏定义——可以用新的定义覆盖描述文件中的宏定义。
- 目标——指定建立描述文件中描述的某个文件，如上面的例子中默认是生成最后的 test.exe 文件，也可以用 nmake x.res 指定更新 x.res 文件。

nmake 常用的选项如表 2.8 所示。

表 2.8 nmake 的常用选项

选 项	简 介
/A	不检测文件时间，强制更新所有文件
/B	文件时间相等时也要更新文件
/D	make 时显示文件新旧信息
/N	显示 make 时要执行的命令，但并不真正执行
/P	一个比较有用的选择，make 时显示详细的信息

由于 nmake 的应用是基于文件时间的，当计算机的时钟不准确或文件拷贝到另一台计算机后文件时间有些偏差，那么文件的更新可能会不正确，这时最好用/A 选项强制把所有文件更新一遍。在平时使用的时候，以 makefile 当做建立的描述文件名，那么仅键入不加参数的 nmake 命令就可以完成所有工作了。

2.4.3 描述文件的语法

`make` 工具最主要也是最基本的功能就是通过描述文件来描述源程序之间的相互关系并自动维护编译工作，而描述文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并链接生成可执行文件，并要求定义源文件之间的依赖关系，为了方便使用，文件中同时可以用一些宏定义。描述文件一般需要包含以下内容：

- 注释
- 宏定义
- 显式规则
- 隐含规则

在这里，首先为 2.4.1 节中有关 `test.exe` 的例子写出一个描述文件，再逐步介绍各部分的书写语法。为了方便使用，一般都把描述文件的文件名取为默认文件名：`makefile`。这个例子的 `makefile` 文件如下（注意前面括号里的是行号，不是文件的真正内容）：

```
(001)  # nmake 工具的描述文件例子
(002)  EXE = Test.exe                      #指定输出文件
(003)  OBJS = x.obj \
(004)         y.obj                        #需要的目标文件
(005)  RES = x.res                         #需要的资源文件
(006)
(007)  LINK_FLAG = /subsystem:windows      #链接选项
(008)  ML_FLAG = /c /coff                 #编译选项
(009)
(010)  #定义依赖关系和执行命令
(011)  $(EXE): $(OBJS) $(RES)
(012)         Link $(LINK_FLAG) /out:$(EXE) $(OBJS) $(RES)
(013)  $(OBJS): Common.inc
(014)  y.obj: y.inc
(015)
(016)  #定义汇编编译和资源编译的默认规则
(017)  .asm.obj:
(018)         ml $(ML_FLAG) $<
(019)  .rc.res:
(020)         rc $<
(021)
(022)  #清除临时文件
(023)  clean:
(024)         del *.obj
(025)         del *.res
```

1. 注释和换行

`makefile` 中的注释是以`#`号开头一直到行尾的字符，当 `nmake` 工具处理到这些字符的时候，它会完全忽略`#`号及其后面的全部字符。

当一行的内容过长的时候，可以用换行符来继续，`makefile` 的换行符是`\`，如例子中的第 3 行和第 4 行可以合并为：

```
OBJS = x.obj y.obj           #需要的目标文件
```

在使用换行符的时候要注意在“`\`”后面不能再加上其他字符，包括注释和空格，否则 `nmake` 检测到“`\`”不在一行的最后，就不会把它当成换行符解释，从而出现错误。

2. 宏定义

`makefile` 中允许使用简单的宏定义指代源文件及其相关编译信息，可以把宏称为变量，在整个描述文件中，只要符合下面语法的行就是宏定义：

```
变量名=变量内容
```

如上面例子文件中的第 2 到第 8 行就是宏定义，在引用宏时只需在变量前加`$`符号，但是要注意的是，如果变量名的长度超过一个字符，在引用时必须加圆括号（`()`），下面都是有效的宏引用：

```
$(LINK_FLAG)
$(EXE)
$A
$(A)
```

其中最后两个引用是完全一致的。

宏定义的使用可以使 `makefile` 的使用更灵活：首先可以使文件便于修改，比如把第 8 行和第 18 行中 `ml` 的选项部分写成宏定义，以后要改变编译选项的时候，只要直接在 `makefile` 文件头部改变宏定义就可以了，不必阅读修改整个 `makefile` 文件；其次，当不止一个地方用到同一个文件的时候，把文件名定义为宏定义可以减少错误，增加可读性，同时也可以便于修改；最大的好处是可以直接在命令行中用新的宏定义覆盖，比如在命令行中键入：

```
nmake ML_FLAG="/c /coff /Fl"
```

那么这时就会以新的`/c /coff /Fl`定义代替 `makefile` 中定义的`/c /coff`，在这种使用中要注意两个问题：一是宏名称要区分大小写，`ML_FLAG` 和 `ml_flag` 是不一样的；二是定义值中有空格的时候要用双引号引起来（没有空格时可以用不用双引号，如 `ML_FLAG=/c`），这使临时使用不同的参数编译文件时可以不修改 `makefile`。

3. 显式规则

`makefile` 中包含有一些规则，这些规则定义了文件之间的依赖关系和产生命令，一个规则的格式是这样的：

```
目标文件：依赖文件：命令           （方法 1）
```

或

目标文件：依赖文件 命令	(方法 2)
-----------------	--------

在规则定义和命令行中，不能包含注释，例子中的第 11 和 12 行把宏定义展开后就是：

```
test.exe: x.obj y.obj x.res
    Link /subsystem:windows /out:test.exe x.obj y.obj x.res
```

这里的目标文件就是 `test.exe`，它依赖于 3 个文件 `x.obj`，`y.obj` 和 `x.res`，如果有必要，产生目标文件的命令就是下面的 `Link` 命令。规则可以用两种方法，用方法 2 的时候，命令可以从第 2 行开始，第 1 行的“；”省略，但是这时命令前面必须有一个 `Tab` 字符，否则 `nmake` 无法区分这究竟是命令还是别的定义。

在同一个规则中，目标文件可以有多个，依赖文件也可以有多个，同时命令也可以由多个命令行组成，当然这时候就必须用第二种方法定义了，否则无法在同一行中写入多条命令。

我们也可以用上例中类似的方法定义其他规则，如 `x.obj` 或 `x.res` 的生成方法，但 `nmake` 如何知道哪个是最终要 `make` 的文件呢？实际上 `nmake` 默认将整个描述文件的第一条规则中的目标文件认为是最终文件，如果我们将第 11，12 行放到第 13 行后面，那么 `x.obj` 和 `y.obj` 的建立规则就成了第一条规则，`nmake` 建立了 `x.obj` 和 `x.obj` 之后就不理会 `test.exe` 的建立了，所以必须把最终需要生成的文件放在第一条规则定义。当然，在 `nmake` 的命令行参数中可以指定要 `make` 的目标，如我们只需生成 `x.res` 文件，那么不必修改 `makefile` 将 `x.res` 的描述规则移动到最前面，而是直接在命令行键入以下命令即可：

```
nmake x.res
```

参数中也可以同时带好几个目标文件名，`nmake` 会一一处理，如果指定的目标文件没有对应的规则，`nmake` 会返回一个出错信息：

```
fatal error U1073: don't know how to make 'xxx 文件'
```

当用户要求 `nmake` 去建造一个目标时，`make` 会去找到这个目标的依赖规则，这时规则中定义的命令并不会立刻被执行，而是首先要做一些事情：`nmake` 先去检查依赖文件是否是另一条规则的目标文件，如果是，则先处理这一条规则；如果不是，`nmake` 再检查各个依赖文件的时间，看这些文件有没有比目标文件更新的，如果没有，`nmake` 会决定不再重新建造目标文件，并给出提示：`'xxx 文件' is up-to-date`，如果依赖文件有比目标文件更新的，才执行命令。

所以一个顺序下来，所有的目标文件，以及它们的依赖文件，以及依赖文件的依赖文件都会被检查并更新，总而言之，一个目标文件的建立包含了顺序正确的指令链接，这个链接结构是树状的，目标文件是根，一级级扩展到多个文件，我们要求的是 `nmake` 去建立链接中处于根部的那个文件，`nmake` 会根据链接结构从目标开始向初始状态前进，最后慢慢回来，在这个过程中执行建立每个文件所必需的命令，一直到最终目标建立完成。

目标也可以没有依赖文件，而且目标也可以不是一个真正存在的文件，如例子第 23 行到第 25 行中的 `clean` 是一个目标，但我们并不是要生成一个 `clean` 文件，而是希望在文件调试完

毕后用 `nmake` 来清除临时文件，当我们键入 `nmake clean` 的时候，工作目录下并没有 `clean` 这个文件，那么 `nmake` 就会去执行 `clean` 定义中的命令，因为 `nmake` 把每一个不存在的目标当做是一个过时的目标，如此一来，就会删除中间过程中的文件 `*.obj` 和 `*.res`。

指出了目标文件全名的规则称为显式规则，但有些类别的文件的编译方法可以是雷同的，如从 `asm` 文件产生 `obj` 文件的命令总是用 `ml`，从 `rc` 文件产生 `res` 文件的命令总是用 `rc`，对于每个文件都写一条规则有些多余，这时候就要用到隐含规则。

4. 隐含规则

隐含规则可以为某一类的文件指出建立的命令，它具体定义了如何将带一个特定扩展名的文件转换成具有另一种扩展名的文件，定义的格式是：

.源扩展名.目标扩展名: ; 命令	(方法 1)
或	
.源扩展名.目标扩展名: 命令	(方法 2)

隐含规则的语法和显式规则相似，也是用 “:” 隔开，在 “;” 下面书写命令，也可以不用 “;” 而将命令写在第 2 行，同理，这时命令之前要加一个 `Tab` 字符。

隐含规则不能有依赖文件，所以 “:” 下面没有内容，例子中的第 17、18 行定义了从 `asm` 文件建立 `obj` 文件的隐含规则，第 19 和 20 行定义了从 `rc` 文件建立 `res` 文件的隐含规则，隐含规则中无法指定确定的输入文件名，因为输入文件名是泛指的同扩展名的一整类文件，这时候就要用到几个特殊的内定宏来指定文件名，这些宏是 `$@`，`$*`，`$?` 和 `$<`，它们的含义如下：

- `$@` —— 全路径的目标文件。
- `$*` —— 除去扩展名的全路径的目标文件。
- `$?` —— 所有源文件名。
- `$<` —— 源文件名（只能用在隐含规则中）。

所以第 19、20 行中的 `rc $<` 用于 `x.rc` 的时候就是 `rc x.rc`，而用于 `y.rc` 时就是 `rc y.rc` 了。

读者可以注意到一些显式规则没有命令行，如第 13 行的 “`$(OBSJ): Common.inc`” 指出了所有的 `obj` 文件都依赖于 `Common.inc` 文件，第 14 行的 “`y.obj: y.inc`” 则指出了 `y.obj` 除了依赖第 13 行的规则外，还依赖于 `y.inc`。但是第 13 行和第 14 行的两条规则都没有指出产生这些 `obj` 文件的命令，所以 `nmake` 处理的时候会到隐含规则中去找命令行，最后会用第 18 行的 “`ml $(ML_FLAG) $<`” 命令去产生这些 `obj` 文件。

2.5 获取资料

对于程序员来说，“高手”和“菜鸟”之间的区别实际上只有两个因素：第一个因素是从事编程时间的长短不同，使经验的多少有所区别；第二个因素就是手头掌握资料的多少了，因为很多问题并不是靠自己钻研可以解决的，必须靠资料，试想在写 `DOS` 汇编程序时如果没有

中断手册，可以自己钻研出来吗？实际上，大部分“菜鸟”向“高手”问的问题完全可以由参考资料解决，即使一个“菜鸟”对某个问题暂时不懂，但手头有解决问题的详细资料，经过一段时间的钻研，问题自然会解决。“高手”就是这样慢慢练成的。

在 Win32 汇编编程中，资料显得尤其重要。在 DOS 时代，整个操作系统的大小不过几十 KB，所有的 BASIC 和 C 命令基本上都可以直接用人脑记忆下来，用于汇编编程的中断手册也基本上可以让人记住常用的部分。

不过，当时钟走到 21 世纪的时候，软件规模飞速膨胀，仅是开发工具就动辄几十 MB，更不用说复杂的操作系统及其他软件了，所以现在完成一件最基本的事情都必须从文档中寻找合适的方法，大部分程序员手边的文档比字典还要厚几倍，并且，在这些浩如烟海的文档中苦苦寻找之后，还不一定能找出一个满意的解答。像 Windows 就是一个数据结构的迷宫，其 API 的资料远比 DOS 下的中断资料要多，在这种情况下，程序员的经验可以在程序的优化和调试方面发挥作用，但如果没有资料，连程序都写不出来，就更谈不到优化和调试了。

在硬件方面，处理器的发展也很快，图书市场上的资料往往要慢一个节拍，国内的图书尤其如此，想了解最新的指令集就必须到厂家的网站上下载最新资料。所以程序员需要一个强大的信息网络来方便信息的查找，方便与软件开发商的交流，特别是通过 Internet。

目前，各大软件开发商都具有各自的程序员信息网络，这些网络能为程序员提供特别的服务和帮助。所以要寻找编程资料，首选方案就是常接触这些网络，如 Borland 公司的 Borland Community，Oracle 公司的 Oracle Technical Network (OTN)，Sun 公司的 Sun Developer，以及 IBM 公司的 DeveloperWorks 等。

2.5.1 Windows 资料的来源

要获取 Windows 的资料自然要到它的老窝——Microsoft 的站点上去，Microsoft 的程序员网络是 MSDN (Microsoft Developers Network)，在这里可以获得微软所有产品和操作系统的相关信息。它的网址是 <http://msdn.microsoft.com>。

MSDN 是一个内容非常全面的信息网络。现在这个网络一共有 300 万注册用户，它不仅在因特网上建立了网站，而且也发行 MSDN 杂志，以及可供订阅的 CD 和 DVD，其中包括编程信息、技术论文、操作系统、文档、工具、程序代码，以及新产品的 Beta 测试包。MSDN 的技术支持方式既有免费信息服务，也有收费的服务，例如，订阅 MSDN 的印刷品，以及购买 MSDN 的 CD 和 DVD 等，购买 MSDN 实际上等于购买了一种服务。

MSDN 的收费服务是以订阅形式出现的一年 4 期的光盘资料库，它有 3 个版本：

- MSDN 开发库：有知识库和一些例子代码，规模为 20 张左右的光盘，可以联机检索，一般可以从这里找到全部的 API 资料、大量的基础知识和代码。
- MSDN 专业版：包括 MSDN 开发库的全部内容，再加上 Microsoft 操作系统类软件，SDK (Software Development Kit) 和 DDK (Device Driver Development Kit)。SDK 和 DDK 是软件开发包和驱动程序开发包，它们包括开发软件或驱动程序的头文件、例子，以及一些开发工具，要想知道一个课题的最佳解决方案就是去看对应的 SDK

和 DDK 中的例子文件。

- MSDN 宇宙版: 包括 MSDN 专业版的内容, 还包括 Microsoft 所有软件, 如 Windows 2000, Office 和 SQL Server 等, 订阅宇宙版可以让用户从一个软件的 Beta 版时就开始评估使用, 从而使开发人员在第一时间内接触到最新的技术与资料, 但这些产品只允许用于测试目的, 不能用于企业环境去架构网络。

MSDN 的订阅费用不菲, 3 个版本每年的订阅费用分别为 1 500 元、8 200 元和 29 000 元, 这显然是一笔不小的费用, 所以很多程序员还是选择在 Microsoft 的站点上使用联机版本, MSDN 站点在内容上显得稍微过繁, 有人对此的评论是: “Microsoft 每次都会提供大量的资料, 以至于可能需要费些力气才能找到所需的东西, 不过这总比什么都不提供强”。

从网上也可以找到 MSDN 的各种独立部分分别下载, 如各种版本的 SDK 和 DDK 等, 当然这不会在 Microsoft 自己的站点上。同时从网上也可以找到一些单独分离出来的帮助文件, 如《Microsoft Win32 Programmer's Reference》, 《Win32 Multimedia Programmer's Reference》, 《OpenGL Programmer's Reference》, 《Windows Sockets2 Application Program Interface》以及其他几乎所有的程序员手册, 它们中间包括了对应的 API 函数的详细资料。

使用这些 Windows 资料时要注意它们几乎全部是以 C 语言的语法提供的, 因为在 Win32 的环境下, 不管是什么语言, 全部都是建立在 Win32 API 的基础上的, 而 Windows 本身就是用 C 开发的。我们要写 Win32 汇编程序, 参考资料也只好使用这些 C 的版本, 这就要求读者对 C 语言中函数的定义、数据类型和数据结构的定义等有基础的了解。也正因如此, 如果读者有用 MFC 编写 Windows 程序的经验, 看完了这本书以后一定会说: “汇编、C、Windows, 怎么是同一回事?” 的确, 在 Win32 环境下, 所有的语言实际上是一回事, 只不过 Visual FoxPro, Visual BASIC 等软件对 API 以及 Windows 的消息体系封装很深, Visual C++ 和 C++ Builder 等软件相对少一点, 而汇编语言则不加任何封装。

有了足够的参考资料以后, 并不代表着就可以用汇编编写出常用的 Win32 程序了, 因为毕竟这些只是金字塔的一个底边而已, 爬上去的路就是学习的过程, 中间最好的参考就是 Win32 汇编的教程和例子, Internet 上有很多的站点是关于 Win32 汇编编程的, 这里列出几个站点, 读者可以从这些站点的链接中找到其他很多的相关站点:

- MASM32 SDK 软件包的官方站点——<http://www.masm32.com>

包括 MASM32 SDK 软件包下载、简单的 Win32 汇编例子和一些网站链接。

- Iczelion 的 Win32 汇编站点——<http://win32asm.cjb.net>

最著名的英文 Win32 汇编站点, 包括 Iczelion 书写的 Win32 汇编教程、大量的例子和一个讨论区。这个网站需要用代理服务器访问。

- 笔者的 Win32 汇编站点——<http://www.win32asm.com.cn>

中文 Win32 汇编站点, 有 Iczelion 汇编教程的中文版以及其他的一些教程, 也包括大量例子, 另外包含本书的勘误表和光盘下载。

2.5.2 Intel 处理器资料

Win32 汇编参考资料另一个重要部分是 Intel 处理器的资料，这些资料大部分可以在 Intel 的官方网站上找到，网址是 <http://www.intel.com>，但 Intel 的网站存在和 Microsoft 的网站同样的问题，就是资料“太多”了反而不容易查找。

Intel 发布的资料大部分是以 PDF 格式出现的，每一类文件有惟一的编号，可以用编号或资料名称从网站的搜索栏中找到对应的 PDF 文件，和 Win32 汇编编程密切相关的是处理器结构和指令集的参考资料——Intel 处理器软件开发员手册《Intel Architecture Software Developer's Manual》，它由 3 个部分组成。

- 第一部分：基本体系（Basic Architecture），编号 24547004；
- 第二部分：指令集参考（Instruction Set Reference），编号 24547104；
- 第三部分：编程指南（System Programming Guide），编号 24547204。

编号中的前缀 245470、245471 和 245472 是文件编号，后面的 04 表示修订版本是第 4 次，读者可以在 Intel 的网站中输入文件编号找到这几个文件并下载使用。它们包括了最新的 MMX 和 SSE 指令的用法。

2.6 构建编程环境

由于 Win32 汇编可以采用多种编译软件，它们的环境设置方法各不相同，对已经入门的读者来说，这不是问题，但初学者往往不能很好地掌握设置的方法，以至于拿到例子程序后编译不出来，不少人往往在例子一而再、再而三编译不出来后深受打击，结果还没有来得及看到汇编的一点影子就“告别”了这个神秘又精彩的世界。

本书的例子是基于 MASM32 SDK 软件包的，本节介绍该软件包使用中的一些问题。

2.6.1 IDE 还是命令行

IDE（Integrated Develop Environment）即集成开发环境，是指在同一个界面中完成从编写源代码到编译，最后到链接的全过程。Microsoft 的 Visual Studio 中的 VC 和 VB 等开发环境就是 IDE 的典型例子，MASM32 SDK 软件包中同样有一个简单的 IDE 环境 Qeditor.exe，这个 IDE 环境实际上只是一个简单的文本编辑器加上一个用户可以自行设置菜单的 Shell，编译链接工作由 Qeditor.exe 在后台调用 Ml.exe 和 Link.exe 等软件完成。

如果要使用这个 IDE 环境，最大的代价就是不得不使用这个简单的编辑器，而一个好的文本编辑器对工作效率的影响是很大的，一个完善的文本编辑器必须包括语法高亮显示、强大的查找替换、无限次 Undo 和 Redo 操作、支持特大型的源文件等功能，MASM32 SDK 中简单的 Qeditor.exe 符合不了这些要求。

所以建议读者还是抛弃这个 IDE 环境，用一个功能强大的文本编辑软件来写源程序，然后在命令行环境中用 nmake 来维护代码，这样有一个额外的好处，就是保存下来的 makefile 文件

记录了文件的编译与链接参数，可以在以后方便维护。

这里介绍两个很适合用来编辑汇编源文件的文本编辑软件：

- EditPlus——这是一个为程序员编写的文本编辑软件，内置 HTML, CSS, PHP, ASP, Perl, C/C++, Java, JavaScript 和 VBScript 的语法高亮显示，也可以下载 ASM 语法文件，它包括了所有文本编辑软件应该具有的功能：自定义工具菜单，显示行号，语法自动完成，列选择功能（以前只在 WPS 中看到过此功能），无限次 Undo/Redo 和语法检查等功能。读者可以从 <http://www.editplus.com> 下载试用版，试用版可以通过输入注册码成为正式版本。
- UltraEdit32——这是一个文本编辑/十六进制编辑软件，实现的功能和 EditPlus 大同小异，另外还有十六进制编辑功能。UltraEdit32 也是一个共享软件，可以从 <http://www.ultraedit.com> 下载，同样可以通过输入注册码成为正式版本。

2.6.2 本书推荐的工作环境

本书建议读者放弃 MASM32 SDK 自带的简单的 IDE 环境，改为在命令行下用 nmake 工具进行代码维护，为了建立这个环境，需要做下面的工作。

第 1 步：安装常用软件，包括编辑软件 Editplus、MSDN、十六进制编辑器 Hexedit、可视化资源编辑器 Resource Workshop、调试工具 Soft-ICE 和反汇编软件 W32DASM 等，如果硬盘空间允许的话，最好安装 Visual C++，以便使用它集成的资源编辑器。

第 2 步：选择一个驱动器安装 MASM32 SDK 软件包，假设软件包安装于 x 盘，那么安装好的目录是 x:\Masm32 目录，对读者来说整个软件包中重要的只有 3 个目录：bin 目录中有汇编编译器 ml.exe，资源编译器 rc.exe 和链接器 Link.exe 等执行文件；include 目录中有各种头文件；lib 目录中有全部导入库。虽然安装文件自动把安装目录名定为 masm32，如果不满意的话，完全可以把这 3 个关键目录拷贝到别的自己命名的目录中，对使用没有任何影响。

第 3 步：建立源文件目录，由于 Win32 汇编不再像 DOS 汇编一样一个项目只有一个 asm 文件，而是包括 asm, rc, makefile 和图标等多个文件，如果把多个项目的文件混在同一个目录中将无法分辨，所以必须为每个项目单独建立一个目录，建议把这些目录集中在一个专门放置源程序的目录中，如 x:\Source 目录。

第 4 步：由于 MASM32 SDK 软件包中没有 nmake.exe 文件，所以要单独寻找 nmake.exe 并拷贝到 bin 目录中。

第 5 步：为这个环境建立一个设置环境变量的批处理文件，假设文件名为 Var.bat，那么这个文件内容如下：

```
@echo off
set include=x:\masm32\Include
set lib=x:\masm32\lib
set path=x:\masm32\bin;%path%
echo on
```

文件中设置了 3 个环境变量：

- `include` 变量指定头文件的搜索目录。定义了这个环境变量后，`Ml.exe` 和 `Rc.exe` 在处理 `asm` 和 `rc` 文件中遇到 `include` 语句时，会自动在环境变量定义的目录中查找 `include` 语句指定的文件，这样 `include` 语句中就不必写头文件的全路径名，如下所示：

<code>include c:\masm32\include\windows.inc</code>	不设置 <code>include</code> 环境变量时的写法
<code>include windows.inc</code>	设置 <code>include</code> 环境变量后可以这样写

这样处理的好处是以后移动了 `MASM32` 的安装位置后不必修改每个源文件中的 `include` 语句。如果使用 `Visual C++` 的集成环境来建立 `rc` 文件的话，为了使 `rc.exe` 能找到头文件，还要把 `VC++` 安装目录下的 `Include` 和 `MFC\Include` 目录包含进来，多个路径之间用 “;” 隔开：

```
set include=x:\masm32\Include;VC 目录\Include;VC 目录\MFC\Include
```

`VC++` 安装目录一般为 `C:\Program Files\Microsoft Visual Studio\VC98\`。

- `lib` 变量指定导入库文件的搜索目录。`Ml.exe` 根据这个变量寻找 `includelib` 语句指定的导入库文件，`Link.exe` 也根据这个变量寻找库文件的位置。设置 `lib` 变量带来的好处同上。
- `path` 变量就不必多解释了。它只是使我们不必在键入命令时带长长的路径而已。

2.6.3 尝试编译第一个程序

按照上面的步骤安装完成后，下面来编译一个程序测试一下。打开一个文件浏览窗口，切换到源文件目录 `x:\Source`，然后把本书所带光盘中的 `Chapter02\Test` 目录拷贝过来，现在有了一个需要编译的 `Test.asm` 文件在 `x:\Source\Test` 目录中。

打开一个 `MS-DOS` 窗口，并键入 `Var` 执行已建立的 `Var.bat`，这时环境变量和路径已经设置好了，可以键入 `SET` 命令验证一下 `include` 和 `path` 等环境串是否正确，然后键入 `x:` 以及 `cd \Source\Test` 切换到要工作的目录中，并键入 `nmake`，当屏幕上出现如下所示的正确的编译链接信息后，`Test.exe` 就建立完成了。

```
Microsoft (R) Program Maintenance Utility   Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.
```

```
ml /c /coff Test.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.
```

```
Assembling: Test.asm
rc Test.rc
Link /subsystem:windows Test.obj Test.res
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

执行一下 `Test.exe`，会出现一个简单的对话框，表示一个可以执行的 `Win32` 程序正确生成了。万事大吉，这样就可以开始 `Win32` 汇编之旅了！

如果编译的过程中出现“can not open file xxx.inc”或“can not open file xxx.lib”的提示时，就要检查 Var.bat 文件中 include 或 lib 的路径是否正确。

下面的工作就是：编辑源程序→切换到 MS-DOS 窗口→键入 nmake 编译→运行生成的可执行文件→切换到文本编辑器修改源程序……如此循环往复调试程序。



测试本书所带光盘中的代码时，将光盘上的代码拷贝到硬盘上后，别忘了把拷贝过来的文件去掉“只读”属性，否则修改或编译时可能会因无法写文件而产生错误。另外，也不要忘了先校对本机的时钟，因为 nmake 工具根据当前时钟和文件时间对比来决定是否执行编译和链接的命令。

执行 Var.bat 设置环境变量的操作只需在每次刚打开 MS-DOS 窗口的时候执行一次就可以了。调试 Win32 汇编程序时常见的桌面如图 2.5 所示：屏幕上一般有个 MS-DOS 窗口来执行编译命令，有一个文本编辑器窗口修改源代码，同时使用 MSDN 等联机帮助文件，当然再开一个 WinAmp 窗口同时听一听 MP3 也不错！

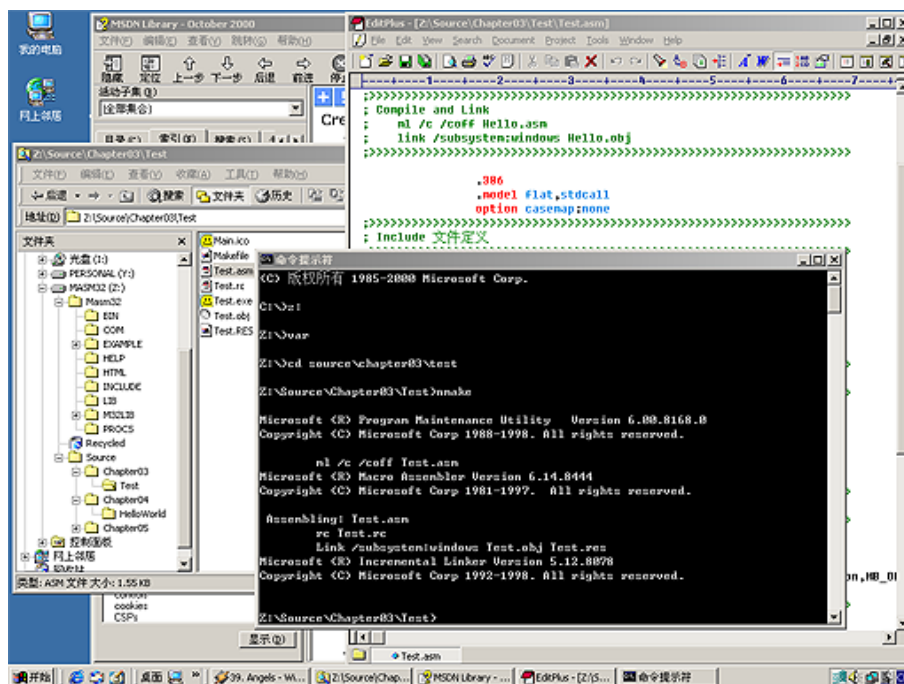


图 2.5 工作环境

第 3 章

使用 MASM

经过第 2 章的准备工作，相信大家都已经建好了 Win32 汇编的工作环境，并已经可以编译程序了，现在让我们来编译所带光盘的 Chapter03\HelloWorld 目录下的“Hello World”程序。这是一个相当小的程序，就和 DOS 时代下经典的“Hello World”程序一样，并没有涉及系统中很多东西，甚至连 Windows 系统中基本的消息驱动机制也没有看到，它只是简单地弹出一个消息框，在上面显示了一句“Hello, World!”，并在文字的下面显示了一个“确定”按钮，就停在那里了，当用户按下“确定”按钮时，程序退出，同时消息框消失。这个程序运行的结果如图 3.1 所示。

但这样一个小程序从结构来看，“麻雀虽小，五脏俱全”，用它来举例说明 Win32 汇编源程序的框架是最合适不过的了，本章将从这个程序出发，探讨 MASM 在 Win32 汇编中的用法，由于篇幅所限，讨论的内容只涉及 MASM 在 Win32 编程中常用的部分。




图 3.1 Win32 汇编的 Hello,World 程序

3.1 Win32 汇编源程序的结构

任何种类的语言，总是有基本的源程序结构规范，在讨论 C 语言的书中，大家都会记得这个非常经典的“Hello World”程序：

```
#include <stdio.h>
main()
{
    printf("Hello, world\n");
}
```

像这样一个程序，就说明了 C 语言中最基本的格式，main() 中的括号和下面的花括号说明了一个函数的定义方法，printf 语句是对函数典型的调用方法，调用函数语句后面的分号也是基本的格式。C 是一种高级语言，在 C 源程序中，不必为堆栈段、数据段和代码段的定义而担心，编译器会把程序中的字符串和语句代码分别放到它们该去的地方，程序开始执行的时候也会自己找到 main() 函数。而汇编是低级语言，必须为所有的语句找到它们该去的地方，所以在



```
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 堆栈段
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
stack                segment stack
                        db          100 dup (?)

stack               ends
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
data                 segment
szHello              db           'Hello, world', 0dh, 0ah, '$'
data                  ends
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 代码段
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
code                 segment
                    assume cs:code, ds:data, ss:stack

start:

                    mov     ax, data
                    mov     ds, ax


                    mov     ah, 9
                    mov     dx, offset szHello
                    int      21h


                    mov     ah, 4ch
                    int      21h

code                 ends
                    end      start
```

到了 Win32 汇编的时候,程序的基本结构还是如此,先来看看这个看起来很新鲜的 Win32 的 Hello World 程序:

[illegible]

怎么样，看来和上面的 C 及 DOS 汇编又不同了吧！但从 `include`、`.data` 和 `.code` 等语句“顾名思义”也能看出一点苗头来，`include` 应该就是包含别的文件，`.data` 想必是数据段，`.code` 应该就是代码段了吧！接下来通过这个例子程序逐段介绍 Win32 汇编程序的结构。

程序的第一部分是模式和源程序格式的定义语句:

这些指令定义了程序使用的指令集、工作模式和格式。

.386 语句是汇编语言的伪指令，它在低版本的宏汇编中就已经存在，类似的指令还有 .8086, .186, .286, .386/.386p, .486/.486p 和 .586/.586p 等，用于告诉编译器在本程序中使用的指令集。在 DOS 的汇编中默认使用的是 8086 指令集，那时候如果在源程序中写入 80386 所特有的指令或使用 32 位的寄存器就会报错，为了在 DOS 环境下进行保护模式编程或仅为了使用 32 位寄存器，常在 DOS 的汇编中使用 .386 来定义。Win32 环境工作在 80386 及以下的处理器中，所以 .386 这一句是必不可少的。

```
mov    cr0, eax
```

. 586

.mmx

2. .model 语句

.model 语句在低版本的宏汇编中已经存在，用来定义程序工作的模式，它的使用方法是：

.model 内存模式[, 语言模式][, 其他模式]

内存模式的定义影响最后生成的可执行文件，可执行文件的规模从小到大，可以有很多种类型，在 DOS 的可执行程序中，有只用到 64 KB 的 .com 文件，也有大大小小的 .exe 文件。到了 Win32 环境下，又有了可以用 4 GB 内存的 PE 格式可执行文件，编写不同类型的可执行文件要用 .model 语句定义不同的参数，具体如表 3.1 所示。

表 3.1 内存模式

模 式	内存使用方式
tiny	用来建立 .com 文件，所有的代码、数据和堆栈都在同一个 64 KB 段内
small	建立代码和数据分别用一个 64 KB 段的 .exe 文件
medium	代码段可以有多个 64 KB 段，数据段只有一个 64 KB 段
compact	代码段只有一个 64 KB 段，数据段可以有多个 64 KB 段
large	代码段和数据段都可以有多个 64 KB 段
huge	同 large，并且数据段中的一个数组也可以超过 64 KB
flat	Win32 程序使用的模式，代码和数据段使用同一个 4 GB 段

在前面章节中已经提到过：Windows 程序运行在保护模式下，系统把每一个 Win32 应用程序都放到分开的虚拟地址空间中去运行，也就是说，每一个应用程序都拥有其相互独立的 4 GB 地址空间，对 Win32 程序来说，只有一种内存模式，即 flat（平坦）模式，意思是内存是很“平坦”地从 0 延伸到 4 GB，再没有 64 KB 段大小限制。对比一下 DOS 版本的 Hello World 和 Win32 版本的 Hello World 开始部分的不同，DOS 程序中有这样两句：

```
mov     ax,data
mov     ds,ax
```

意思是把数据段寄存器 DS 指向 data 数据段，data 数据段在前面已经用 data segment 语句定义，只要 DS 不重新设置，那么从此以后指令中涉及的数据默认将从 data 数据段中取得，所以下面的语句是从 data 数据段取出 szHello 字符串的地址后再显示：

```
mov     ah,9
mov     dx,offset szHello
int     21h
```

纵观 Win32 汇编的源程序，没有一处可以找到 ds 或 es 等段寄存器的使用，因为所有的 4 GB 空间用 32 位的寄存器全部都能访问到了，不必在头脑中随时记着当前使用的是哪个数据段，这就是“平坦”内存模式带来的好处。

如果定义了.model flat，MASM 自动为各种段寄存器做了如下定义：

```
ASSUME  cs:FLAT, ds:FLAT, ss:FLAT, es:FLAT, fs:ERROR, gs:ERROR
```

也就是说, CS, DS, ES 和 SS 段全部使用平坦模式, FS 和 GS 寄存器默认不使用, 这时若在源程序中使用 FS 或 GS, 在编译时会报错。如果有必要使用它们, 只需在使用前用下面的语句声明一下就可以了(比如在第 14 章的异常处理中要用到 FS 寄存器):

```
assume    fs:nothing, gs:nothing 或者 assume fs:flat, gs:flat
```

在 Win32 汇编中, .model 语句中还应该指定语言模式, 即子程序的调用方式, 例子中用的是 stdcall, 它指出了调用子程序或 Win32 API 时参数传递的次序和堆栈平衡的方法, 相对于 stdcall, 不同的语言类型还有 C, SysCall, BASIC, FORTRAN 和 PASCAL, 虽然各种高级语言在调用子程序时都是使用堆栈来传递参数, 但它们的处理方法各有不同。要和其他语言配合, 就必须指定相应的语言种类。Windows 的 API 调用使用的是 stdcall 格式, 所以在 Win32 汇编中没有选择, 必须在 .model 中加上 stdcall 参数。关于参数传递的细节, 在 3.4.2 节中有详细的描述。

3. option 语句

用 option 语句定义的选项有很多, 如 option language 定义和 option segment 定义等, 在 Win32 汇编程序中, 需要的只是定义 option casemap:none, 这个语句定义了程序中的变量和子程序名是否对大小写敏感, 由于 Win32 API 中的 API 名称是区分大小写的, 所以必须指定这个选项, 否则在调用 API 的时候会有问题。

3.1.2 段的定义

1. 段的概念

把上面的 Win32 的 Hello World 源程序中的语句归纳精简一下, 再列在下面:

```
.386
.model flat, stdcall
option casemap:none
<一些 include 语句>
.data
<一些字符串、变量定义>
.code
<代码>
<开始标号>
<其他语句>
end 开始标号
```

上一节讲到的选项、模式等定义并不会在编译好的可执行程序中产生什么, 它们只是“说明”, 而真正的数据和代码是定义在各个段中的, 如上面的 .data 段和 .code 段, 考虑到不同的数据类型, 还可以有其他种类的数据段, 下面是包含全部段的源程序结构:

```
.386
.model flat, stdcall
option casemap:none
<一些 include 语句>
.stack [堆栈段的大小]
.data
```

```
<一些初始化过的变量定义>
.data?
<一些没有初始化过的变量定义>
.const
<一些常量定义>
.code
<代码>
<开始标号>
<其他语句>
end 开始标号
```

`.stack`, `.data`, `.data?`, `.const` 和 `.code` 是分段伪指令, Win32 中实际上只有代码和数据之分, `.data`, `.data?` 和 `.const` 都是数据段, `.code` 是代码段, 与 DOS 汇编不同, 由于 Win32 汇编不必考虑堆栈, 系统会为程序分配一个向下扩展的、足够大的段作为堆栈段, 所以 `.stack` 段定义常常被忽略。



前面不是说过 Win32 环境下不用“段”了吗? 是的, 这些“段”实际上并不是 DOS 汇编中那种意义的段, 而是内存的“分段”。上一个段的结束就是下一个段的开始, 所有的“分段”合起来, 包括系统使用的地址空间, 就组成了整个可以寻址的 4 GB 空间。由于 Win32 环境的内存管理使用了 80386 处理器的分页机制, 每个页 (4 KB 大小) 可以自由指定属性, 所以上一个 4 KB 可能是代码, 属性是可执行但不可写, 下一个 4 KB 就有可能是既可读也可写但不可执行的数据, 再下面呢? 有可能是可读不可写也不可执行的数据。Win32 汇编源程序中“分段”的概念实际上是把不同类型的数据或代码归类, 再放到不同属性的内存页 (也就是不同的“分段”) 中, 这中间不涉及使用不同的段选择器。虽然使用和 DOS 汇编同样的 `.code` 和 `.data` 语句来定义, 意思可是完全不同了! 为了简单起见, 在本书中还是简称“段”, 读者应该注意到其中不同的含义。

2. 数据段

`.data`, `.data?` 和 `.const` 定义的是数据段, 分别对应不同方式的数据定义, 在最后生成的可执行文件中也分别放在不同的节区 (Section) 中。程序中的数据定义一般可以归纳为 3 类。

第一类是可读可写的已定义变量。这些数据在源程序中已经被定义了初始值, 而且在程序的执行中有可能被更改, 如一些标志等, 这些数据必须定义在 `.data` 段中, `.data` 段是已初始化数据段, 其中定义的数据是可读可写的, 在程序装入完成的时候, 这些值就已经在内存中了, `.data` 段一般存放在可执行文件的 `_DATA` 节区内。

第二类是可读可写的未定义变量。这些变量一般是当做缓冲区或者在程序执行后才开始使用的, 这些数据可以定义在 `.data` 段中, 也可以定义在 `.data?` 段中, 但一般把它放到 `.data?` 段中。虽然定义在这两种段中都可以正常使用, 但定义在 `.data?` 段中不会增大 `.exe` 文件的大小。举例说明, 如果要用到一个 100 KB 的缓冲区, 可以用下面的语句定义:

<code>szBuffer</code>	<code>db</code>	<code>100 * 1024 dup (?)</code>
-----------------------	-----------------	---------------------------------

这个语句如果放在 `.data` 段中, 编译器认为这些数据在程序装入时必须有效, 所以它在

生成可执行文件的时候保留了所有的 100 KB 的内容,即使它们是全零!假设程序其他部分的大小是 50 KB,那么最后的 .exe 文件就会是 150 KB 大小,如果缓冲区定义为 1 MB,那么 .exe 文件会增大到 1 050 KB。.data?段则不同,其中的内容编译器会认为程序在开始执行后才会用到,所以在生成可执行文件的时候只保留了大小信息,不会为它浪费磁盘空间。在与上面同样的情况下,即使缓冲区定义为 1 MB,可执行文件同样只有 50 KB!总之,.data?段是未初始化数据段,其中的数据也是可读可写的,但在可执行文件中不占空间,.data?段在可执行文件中一般存放在_BSS 节区中。

第三类数据是一些常量。如一些要显示的字符串信息,它们在程序装入的时候也已经有效,但在整个执行过程中不需要修改,这些数据可以放在 .const 段中,.const 段是常量段,它是可读不可写的。为了方便起见,在小程序中常常把常量一起定义到 .data 段中,而不另外定义一个 .const 段。在程序中如果不小心用了对 .const 段中的数据做写操作的指令,会引起保护错误,Windows 会显示一个如图 3.2 所示的提示框并结束程序。

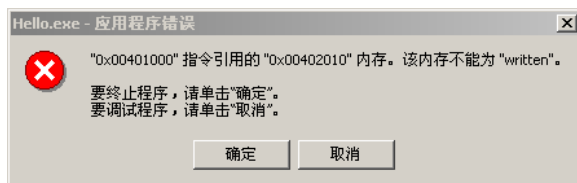


图 3.2 对 .const 段写操作引起的非法操作

如果不怕程序可读性不佳的话,把 .const 段中定义的内容混到 .code 段中去也可以正常使用,因为 .code 段也是可以读的。

3. 代码段

.code 段是代码段,所有的指令都必须写在代码段中,在可执行文件中,代码段一般是放在 _TEXT 节区中的。Win32 环境中的数据段是不可执行的,只有代码段有可执行的属性。对于工作在特权级 3 的应用程序来说,.code 段是不可写的,在编 DOS 汇编程序的时候,好事的程序员往往有个习惯,就是靠改动代码段中的代码来做一些反跟踪的事情,如果企图在 Win32 汇编下做同样的事情,结果就是和上面同样的“非法操作”。

当然事物总有两面性,在 Windows 95 下,在特权级 0 下运行的程序对所有的段都有读写的权利,包括代码段。另外,在优先级 3 下运行的程序也不是一定不能写代码段,代码段的属性是由可执行文件 PE 头部中的属性位决定的,通过编辑磁盘上的 .exe 文件,把代码段属性位改成可写,那么在程序中就允许修改自己的代码段。一个典型的应用就是一些针对可执行文件的压缩软件和加壳软件,如 Upx 和 PeCompact 等,这些软件靠把代码段进行变换来达到解压缩或解密的目的,被处理过的可执行文件在执行时需要由解压代码来将代码段解压缩,这就需要写代码段,所以这些软件对可执行文件代码段的属性预先做了修改。

4. 堆栈段

在程序中不必定义堆栈段,系统会自动分配堆栈空间。惟一值得一提的是,堆栈段的内存属性是可读写并且是可执行的,这样靠动态修改代码的反跟踪模块可以拷贝到堆栈中去边修改

边执行。一些病毒或者黑客工具用到的缓冲区溢出技术也用到了这个特征，有兴趣了解的读者可以查阅相关的资料。

3.1.3 程序结束和程序入口

在C语言源程序中，程序不必显式地指定程序由哪里开始执行，编译器已经约定好从main()函数开始执行了。而在汇编源程序中，并没有一个main函数，程序员可以指定从代码段的任何一个地方开始执行，这个地方由程序最后一句的end语句来指定：

```
end      [开始地址]
```

这句语句同时表示源程序结束，所有的代码必须在end语句之前，例如：

```
end      start
```

上述语句指定程序从start这个标号开始执行。当然，start标号必须在程序的代码段中有所定义。

但是，一个源程序不必非要指定入口标号，这时候可以把开始地址忽略不写，这种情况发生在编写多模块程序的单个模块的时候。当分开写多个程序模块时，每个模块的源程序中也可以包括.data、.data?、.const和.code段，结构就和上面的Win32 Hello World一样，只是其他模块最后的end语句必须不带开始地址。当最后把多个模块链接在一起的时候，只能有一个主模块指定入口地址，在多个模块中指定入口地址或者没有一个模块指定了入口地址，链接程序都会报错。

3.1.4 注释和换行

注释是源程序中不可忽略的一部分，汇编源程序的注释以分号(;)开始，注释既可以在一行的头部，也可以在一行的中间，一行中所有在分号之后的字符全部当做注释处理，但在字符串的定义中包含在引号内的分号不当做是注释的开始。

举例如下：

```
;这里是注释
call _    PrintChar          ;这里是注释
szChar    db 'Hello, world;',0dh,0ah ;world后面的分号不是注释，后面的才是
```

当源程序的某一行过长，不利于阅读的时候，可以分行书写，分行的办法是在一行的最后用反斜杠(\)做换行符，如：

```
invoke    MessageBox, NULL, offset szText, offset szCaption, MB_OK
```

可以写为：

```
invoke    MessageBox, \
          Null, \           ;父窗口句柄
          offset szText, \   ;消息框中的文字
          offset szCaption, \ ;标题文字
          MB_OK
```

“一行的最后”指的是最后一个有用的字符，反斜杠后面多几个空格或加上注释并不影响换行符的使用，如上例所示，这一点与 makefile 文件中换行符的规定有所不同。

3.2 调用 API

3.2.1 API 是什么

Win32 程序是构筑在 Win32 API 基础上的。在 Win32 API 中，包括了大量的函数、结构和消息等，它不仅为应用程序所调用，也是 Windows 自身的一部分，Windows 自身的运行也调用这些 API 函数。

在 DOS 下，操作系统的功能是通过各种软中断来实现的，如大家都知道 int 21h 是 DOS 中断，int 13h 和 int 10h 是 BIOS 中的磁盘中断和视频中断。当应用程序要引用系统功能时，要把相应的参数放在各个寄存器中再调用相应的中断，程序控制权转到中断中去执行，完成以后会通过 iret 中断返回指令回到应用程序中。如 DOS 汇编下的 Hello World 程序中有下列语句：

```
mov     ah,9
mov     dx,offset szHello
int     21h
```

这 3 条语句调用 DOS 系统模块中的屏幕显示功能，功能号放在 ah 中，9 号功能表示屏幕显示，要输出到屏幕上的内容的地址放在 dx 中，然后去调用 int 21h，字符串就会显示到屏幕上。

这个例子说明了应用程序调用系统功能的一般过程。首先，系统提供功能模块并约定参数的定义方法，同时约定调用的方式，应用程序按照这个约定来调用系统功能。在这里，ah 中放功能号 9，dx 中放字符串地址就是约定的参数，int 21h 是约定的调用方式。

下面来看看这种方法的不便之处。首先，所有的功能号定义是冷冰冰的数字，int 21h 的说明文档是这样的：

```
Int 21 Functions:

00—Program termination
01—Keyboard input
02—Display output
03—AUX input
04—AUX output
05—Printer output
06—Direct console I/O
07—Direct STDIN input, no echo
08—Keyboard input, no echo
09—Print string
0A—Buffered keyboard input
0B—Check standard input status
```

再进入 09 号功能看使用方法：

```
Print string (Func 09)
```

```
AH = 09h
DS:DX -> string terminated by "$"
```

这就是 DOS 时代汇编程序员都有一厚本《中断大全》的原因，因为所有的功能编号包括使用的参数定义仅从字面上看，是看不出一点头绪来的。

另外，80x86 系列处理器能处理的中断最多只能有 256 个，不同的系统服务程序使用了不同的中断号，这可怜的中断数量显得太少了，结果到最后是中断挂中断，大家抢来抢去的，把好好的一个系统搞得像接力赛跑一样。

对于这些弱点，程序员们都有个愿望：系统功能如果能以功能名称作为子程序名直接调用就好了，参数也最好定义得有意义一点，这样一来写程序就会方便得多，编系统扩展模块就不必总是操心往哪个中断上面挂了，最好能把上面 `int 21h/ah=9` 的调用写成下面这副样子：

```
call    PrintString,addr szHello
```

终于，好消息出来了，Win32 环境中的编程接口就是这个样子！这就是 API，它实际上是以一种新的方法代替了 DOS 中用软中断的方式。与 DOS 的结构相比，Win32 的系统功能模块放在 Windows 的动态链接库（DLL）中，DLL 是一种 Windows 的可执行文件，采用的是和 .exe 文件同样的 PE 格式，在 PE 格式文件头的导出表中，以字符串形式指出了这个 DLL 能提供的函数列表。应用程序使用字符串类型的函数名指定要调用的函数。

应用程序在使用的时候由 Windows 自动装入 DLL 程序并调用相应的函数。

实际上，Win32 的基础就是由 DLL 组成的。Win32 API 的核心由 3 个 DLL 提供，它们是：

- KERNEL32.DLL——系统服务功能。包括内存管理、任务管理和动态链接等。
- GDI32.DLL——图形设备接口。利用 VGA 与 DRV 之类的显示设备驱动程序完成显示文本和矩形等功能。
- USER32.DLL——用户接口服务。建立窗口和传送消息等。

当然，Win32 API 还包括其他很多函数，这些也是由 DLL 提供的，不同的 DLL 提供了不同的系统功能。如使用 TCP/IP 协议进行网络通信的 DLL 是 `Wsock32.dll`，它所提供的 API 称为 Socket API；专用于电话服务方面的 API 称为 TAPI（Telephony API），包含在 `Tapi32.dll` 中。所有的这些 DLL 提供的函数组成了现在所用的 Win32 编程环境。

3.2.2 调用 API

与在 DOS 中用中断方式调用系统功能一样，用 API 方式调用存放在 DLL 中的函数必须同样约定一个规范，用来定义函数的调用方法、参数的传递方法和参数的定义，洋洋洒洒几百 MB 的 Windows 系统比起才几百 KB 规模的 DOS，其系统函数的规模和复杂程度都上了一个数量级，所以在使用一个 API 时，带的参数数量多达十几个是常有的事，在 DOS 下用寄存器来传递参数的方法显然已经不能胜任了。

Win32 API 是用堆栈来传递参数的，调用者把参数一个个压入堆栈，DLL 中的函数程序再从堆栈中取出参数处理，并在返回之前将堆栈中已经无用的参数丢弃。在 Microsoft 发布的《Microsoft Win32 Programmer's Reference》中定义了常用 API 的参数和函数声明，先来看消息框函数的声明：

```
int MessageBox(  
    HWND hWnd,           // handle to owner window  
    LPCTSTR lpText,       // text in message box  
    LPCTSTR lpCaption,    // message box title  
    UINT uType            // message box style  
);
```

最后还有一句说明：

Library: Use User32.lib.

上述函数声明说明了 MessageBox 有 4 个参数，它们分别是 HWND 类型的窗口句柄 (hWnd)，LPCTSTR 类型的要显示的字符串地址 (lpText) 和标题字符串地址 (lpCaption)，还有 UINT 类型的消息框类型 (uType)。这些数据类型看起来很复杂，但有一点是很重要的，对于汇编语言来说，Win32 环境中的参数实际上只有一种类型，那就是一个 32 位的整数，所有这些 HWND, LPCTSTR 和 UINT 实际上就是汇编中的 dword (double word)，之所以定义为不同的模样，是为了说明其用途。可能是因为 Windows 是用 C 写成的吧，或者是因为世界上的程序员用 C 语言的最多，Windows 所有编程资料发布的格式都是用 C 格式的。

上面的声明用汇编的格式来表达就是：

MessageBox Proto hWnd:dword, lpText:dword, lpCaption:dword, uType:dword

上面最后一句 Library: Use User32.lib 则说明了这个函数包括在 User32.dll 中。

有了函数原型的定义以后，就是调用的问题了，Win32 API 调用中要把参数放入堆栈，顺序是最后一个参数最先进栈，在汇编中调用 MessageBox 函数的方法是：

```
push    uType  
push    lpCaption  
push    lpText  
push    hWnd  
call    MessageBox
```

在源程序编译链接成可执行文件后，call MessageBox 语句中的 MessageBox 会被换成一个

地址，指向可执行文件中的导入表，导入表中指向 MessageBox 函数的实际地址会在程序装入内存的时候，根据 User32.dll 在内存中的位置由 Windows 系统动态填入。

1. 使用 invoke 语句

API 是可以调用了，另一个烦人的问题又出现了，Win32 的 API 动辄就是十几个参数，整个源程序一眼看上去基本上都是把参数压入堆栈的 push 指令，参数的个数和顺序很容易搞错，由此引起的莫名其妙的错误源源不断，源程序的可读性看上去也很差。如果写的时候少写了一句 push 指令，程序在编译和链接的时候都不会报错，但在执行的时候必定会崩溃，原因是堆栈对不齐了。

有没有解决的办法呢？最好是像 C 语言一样，能在同一句中打入所有的参数，并在参数使用错误的时候能够提示。

Microsoft 终于做了一件好事，在 MASM 中提供了一个伪指令实现了这个功能，那就是 invoke 伪指令，它的格式是：

```
invoke  函数名[, 参数 1][, 参数 2].....
```

对 MessageBox 的调用在 MASM 中可以写成：

```
invoke  MessageBox, NULL, offset szText, offset szCaption, MB_OK
```

注意，invoke 并不是 80386 处理器的指令，而是一个 MASM 编译器的伪指令，在编译的时候由编译器把上面的指令展开成我们需要的 4 个 push 指令和 1 个 call 指令，同时，进行参数数量的检查工作，如果带的参数数量和声明时的数量不符，编译器会报错：

```
error A2137: too few arguments to INVOKE
```

编译时看到这样的错误报告，首先要检查的是有没有少写了一个参数。由于对于不带参数的 API 调用，invoke 伪指令的参数检查功能可有可无，所以既可以用 call API_Name 这样的语法也可以用 invoke API_Name 这样的语法。



TASM 中没有 invoke 伪指令，它直接使用 call 指令实现同样的功能，如在 TASM 源代码中写上：

```
call  MessageBox, NULL, offset szText, offset szCaption, MB_OK
```

TASM 编译器会将其展开成我们需要的 4 个 push 指令和 1 个 call 指令。

2. API 函数的返回值

有的 API 函数有返回值，如 MessageBox 定义的返回值是 int 类型的数，返回值的类型对汇编程序来说也只有 dword 一种类型，它永远放在 eax 中。如果要返回的内容不是一个 eax 所能容纳的，Win32 API 采用的方法一般是 eax 中返回一个指向返回数据的指针，或者在调用参数中提供一个缓冲区地址，干脆把数据直接返回到缓冲区中去。

3. 函数的声明

在调用 API 函数的时候，函数原型也必须预先声明，否则，编译器会不认这个函数。invoke

伪指令也无法检查参数个数。声明函数的格式是：

函数名 proto [距离] [语言] [参数 1]:数据类型, [参数 2]:数据类型,

句中的 proto 是函数声明的伪指令，距离可以是 NEAR, FAR, NEAR16, NEAR32, FAR16 或 FAR32, Win32 中只有一个平坦的段，无所谓距离，所以在定义时是忽略的；语言类型就是 .model 那些类型，如果忽略，则使用 .model 定义的默认值。

后面就是参数的列表了，由于 Win32 API 仅仅使用 dword 类型的参数，所以绝大多数的数据类型都是 dword，另外对于编译器来说，它只关心参数的数量，参数的名称在这里是“无用”的，仅是为了可读性而设置的，可以省略掉，所以下面两句消息框函数的定义实际上是一样的：

MessageBox Proto hWnd:dword, lpText:dword, lpCaption:dword, uType:dword
 MessageBox Proto :dword, :dword, :dword, :dword

在 Win32 环境中，和字符串相关的 API 共有两类，分别对应两个字符集：一类是处理 ANSI 字符集的，另一类是处理 Unicode 字符集的。前一类函数名字的尾部带一个“A”字符，处理 Unicode 的则带一个“W”字符。我们比较熟悉的 ANSI 字符串是以 NULL 结尾的一串字符数组，每一个 ANSI 字符占一个字节宽。对于欧洲语言体系，ANSI 字符集已经足够了，但对于有成千上万个不同字符的几种东方语言体系来说，Unicode 字符集更有用。每一个 Unicode 字符占两个字节的宽度，这样一来就可以同时定义 65 536 个不同的字符了。

MessageBox 和显示字符串有关，同样它有两个版本，严格地说，系统中有两个定义：

MessageBoxA Proto hWnd:dword, lpText:dword, lpCaption:dword, uType:dword
 MessageBoxW Proto hWnd:dword, lpText:dword, lpCaption:dword, uType:dword

虽然《Microsoft Win32 Programmer's Reference》中只有一个 MessageBox 定义，从 MSDN 上查询也是如此，但 User32.dll 中确实没有 MessageBox，而只有 MessageBoxA 和 MessageBoxW，那么为什么在源代码中还是可以使用 MessageBox 呢？这是因为在程序的头文件 user32.inc 中有一句：

MessageBox equ <MessageBoxA>

它把 MessageBox 偷梁换柱变成了 MessageBoxA。在源程序中继续沿用 MessageBox 是为了程序的可读性，以及保持和手册的一致性，但对于编译器来说，实际是在使用 MessageBoxA。

并不是每个 Win32 系统都支持 W 系列的 API，在 Windows 9x 系列中，对 Unicode 是不支持的，绝大多数的 API 只有 ANSI 版本（为数不多的几个例外是存在 MessageBoxW 等函数，它们让 Unicode 版本的程序在检测到系统不支持的时候，能有机会用消息框提示用户并退出），只有 Windows NT 系列才对 Unicode 完全支持。为了编写在几个平台中通用的程序，一般应用程序都使用 ANSI 版本的 API 函数集。



为了使程序更有移植性，在源程序中一般不直接指明使用 Unicode 还是 ANSI 版本，而是使用宏汇编中的条件汇编功能来统一替换，如在源程序中使用 MessageBox，但在头文件中定义：

```
if      UNICODE
    MessageBox     equ     <MessageBoxW>
else
    MessageBox     equ     <MessageBoxA>
endif
```

所有涉及版本问题的 API 都可以按此方法定义，然后在源程序的头部指定 UNICODE=1 或 UNICODE=0，重新编译后就能产生不同的版本。

4. include 语句

对于所有要用到的 API 函数，在程序的开始部分都必须预先声明，但这个步骤显然是比较麻烦的，为了简化操作，可以采用各种语言通用的解决办法，就是把所有的声明预先放在一个文件中，在用到的时候再用 include 语句包含进来。现在回到 Win32 Hello World 程序，这个程序用到了两个 API 函数：MessageBox 和 ExitProcess，它们分别在 User32.dll 和 Kernel32.dll 中，在 MASM32 SDK 软件包中已经包括了所有 DLL 的 API 函数声明列表，每个 DLL 对应<DLL 名.inc>文件，在源程序中只要使用 include 语句包含进来就可以了：

```
include     user32.inc
include     kernel32.inc
```

当用到其他的 API 函数时，只需相应增加对应的 include 语句。

include 语句还用来在源程序中包含其他文件，当多个源程序用到相同的函数定义、常量定义，甚至源代码时，可以把相同的部分写成一个文件，然后在不同的源程序中用 include 语句包含进来。

编译器对 include 语句的处理仅是简单地把这一行用指定的文件内容替换掉而已。

include 语句的语法是：

```
include     文件名
或          include     <文件名>
```

当遇到要包括的文件名和 MASM 的关键字同名等可能会引起编译器混淆的情况时，可以用“< >”将文件名括起来。

5. includelib 语句

在 DOS 汇编中，使用中断调用系统功能是不必声明的，处理器自己知道到中断向量表中去取中断地址。在 Win32 汇编中使用 API 函数，程序必须知道调用的 API 函数存在于哪个 DLL 中，否则，操作系统必须搜索系统中存在的所有 DLL，并且无法处理不同 DLL 中的同名函数，这显然是不现实的，所以，必须有个文件包括 DLL 库正确的定位信息，这个任务是由导入库来实现的。

在使用外部函数的时候，DOS 下有函数库的概念，那时的函数库实际上是静态库，静态库

是一组已经编写好的代码模块，在程序中可以自由引用，在源程序编译成目标文件，最后要链接成可执行文件的时候，由 link 程序从库中找出相应的函数代码，一起链接到最后的可执行文件中。DOS 下 C 语言的函数库就是典型的静态库。库的出现为程序员节省了大量的开发时间，缺点就是每个可执行文件中都包括了要用到的相同函数的代码，占用了大量的磁盘空间，在执行的时候，这些代码同样重复占用了宝贵的内存。

Win32 环境中，程序链接的时候仍然要使用函数库来定位函数信息，只不过由于函数代码放在 DLL 文件中，库文件中只留有函数的定位信息和参数数目等简单信息，这种库文件叫做导入库，一个 DLL 文件对应一个导入库，如 User32.dll 文件用于编程的导入库是 User32.lib，MASM32 SDK 软件包中包含了所有 DLL 的导入库。

为了告诉链接程序使用哪个导入库，使用的语句是：

	includelib	库文件名
或	includelib	<库文件名>

与 include 的用法一样，在要包括让编译器混淆的文件名时加方括号。Win32 Hello world 程序用到的两个 API 函数 MessageBox 和 ExitProcess 分别在 User32.dll 和 Kernel32.dll 中，那么在源程序使用的相应语句为：

includelib	user32.lib
includelib	kernel32.lib

和 include 语句的处理不同，includelib 不会把 .lib 文件插入到源程序中，它只是告诉链接器在链接的时候到指定的库文件中去找 API 函数的位置信息而已。

3.2.3 API 参数中的等值定义

再回过头来看显示消息框的语句：

invoke	MessageBox, NULL, offset szText, offset szCaption, MB_OK
--------	--

在 uType 这个参数中使用了 MB_OK，这个 MB_OK 是什么意思呢，先来看《Microsoft Win32 Programmer's Reference》中的说明：

uType —— 定义对话框的类型，这个参数可以是以下标志的合集：

要定义消息框上显示按钮，用下面的某一个标志：

MB_ABORTRETRYIGNORE —— 消息框有三个按钮：“终止”，“重试”和“忽略”

MB_HELP —— 消息框上显示一个“帮助”按钮，按下后发送 WM_HELP 消息

MB_OK —— 消息框上显示一个“确定”按钮，这是默认值

MB_OKCANCEL —— 消息框上显示两个按钮：“确定”和“取消”

MB_RETRYCANCEL —— 消息框上显示两个按钮：“重试”和“忽略”

MB_YESNO —— 消息框上显示两个按钮：“是”和“否”

MB_YESNOCANCEL —— 消息框上显示三个按钮：“是”、“否”和“取消”

要在消息框中显示图标，用下面的某一个标志：

MB_ICONWARNING —— 显示惊叹号图标

MB_ICONINFORMATION —— 显示消息图标

MB_ICONASTERISK —— 显示危险图标

MB_ICONQUESTION —— 显示问号图标

MB_ICONSTOP —— 显示停止图标

.....

这些是 uType 参数说明中的一小半，可以看出，参数中可以用的值有很多种，让我们换一个值试试看，把语句改为：

```
invoke    MessageBox, NULL, offset szText, \
          offset szCaption, MB_ICONWARNING or MB_YESNO
```

再编译执行看，屏幕上出现了一个不一样的消息框，如图 3.3 所示。

和参数说明中的一样！消息框中出现了一个惊叹号图标，按钮也变成了“是”和“否”两个按钮！MB_ICONWARNING 和 MB_YESNO 等参数究竟是什么意思呢，MASM 中显然没有这样的预定义，让我们先来找 Visual C++ 的头文件，在 WinUser.h 中可以找到下面一段：



图 3.3 另一个消息框

```
/*
 * MessageBox() Flags
 */
#define MB_OK                0x00000000L
#define MB_OKCANCEL          0x00000001L
#define MB_ABORTRETRYIGNORE  0x00000002L
#define MB_YESNOCANCEL       0x00000003L
#define MB_YESNO             0x00000004L
#define MB_RETRYCANCEL       0x00000005L

#define MB_ICONHAND          0x00000010L
#define MB_ICONQUESTION      0x00000020L
#define MB_ICONEXCLAMATION   0x00000030L
#define MB_ICONASTERISK      0x00000040L

#if(WINVER >= 0x0400)
#define MB_USERICON          0x00000080L
#define MB_ICONWARNING       MB_ICONEXCLAMATION
#define MB_ICONERROR         MB_ICONHAND
#endif /* WINVER >= 0x0400 */

#define MB_ICONINFORMATION    MB_ICONASTERISK
#define MB_ICONSTOP           MB_ICONHAND
.....
```

显然，MB_YESNO 就是 4，MB_ICONWARNING 就是 30h，默认的 MB_OK 就是 0，Win32 API 的参数使用这样的定义方法是为了免除程序员死记数值定义的麻烦。在编写 Win32 汇编程序时，MASM32 SDK 软件包中的 Windows.inc 也包括了所有这些参数的定义，只要在程序的开头包含这个定义文件：

```
include    windows.inc
```

就可以方便地完全按照 API 手册来使用 Win32 函数。

打开\masm32\include 目录下的 Windows.inc 查看一下,可以发现整个文件总共有两万六千多行,包括了几乎所有的 Win32 API 参数中的常量和数据结构定义。正是有了这个文件中详尽的定义,Win32ASM 才得以流行起来,试想一下,哪个程序员愿意每使用一个 API 语句,就到函数手册中去看参数,然后到 Microsoft 发布的 Visual C++ 的头文件中去找对应的数值,再应用到汇编源程序中?这样会有 80% 以上的时间花在做无用功上(最后还是要骂 Microsoft 为什么不提供汇编格式的头文件,毕竟 MASM32 SDK 软件包不是 Microsoft 出的)。

有时候由于版本的原因,当使用最新的 API 手册时,会发现有些参数使用的常量在 Windows.inc 中并没有定义,这下惨了,谁都不知道类似于 MB_XXXXYY 的符号代表什么数值,Microsoft 的《Microsoft Programmer's Reference》手册中从来就不会把参数对应的数值写进去。遇到这种情况,只有拿出最原始的办法了,就是到最新的 Visual C++ 或 SDK 的 include 目录中去,在 C 语言格式的 .h 头文件中把定义找出来,然后自行增补到 Windows.inc 中去。如果这样也找不到定义值的话,那只好放弃使用这个 API 了。

3.3 标号、变量和数据结构

当程序中要跳转到另一位置时,需要有一个标识来指示新的位置,这就是标号,通过在目标地址的前面放上一个标号,可以在指令中使用标号来代替直接使用地址。

使用变量是任何编程语言都要遇到的工作,Win32 汇编也不例外,在 MASM 中使用变量也有需要注意的几个问题,错误地使用变量定义或用错误的方法初始化变量会带来难以定位的错误。变量是计算机内存中已命名的存储位置,在大部分的语言中都有很多种类的变量,如整数型、浮点型和字符串等,不同的变量有不同的用途和尺寸,比如说虽然长整数和单精度浮点数都是 32 位长,但它们的用途不同。

顾名思义,变量的值在程序运行中是需要改变的,所以它必须定义在可写的段内,如 .data 和 .data?, 或者在堆栈内。按照定义的位置不同,MAASM 中的变量也分为全局变量和局部变量两种。

在 MASM 中标号和变量的命名规范是相同的,它们是:

- (1) 可以用字母、数字、下划线及符号@、\$和?。
- (2) 第一个符号不能是数字。
- (3) 长度不能超过 240 个字符。
- (4) 不能使用指令名等关键字。
- (5) 在作用域内必须是惟一的。

3.3.1 标号

1. 标号的定义

当在程序中使用一条跳转指令的时候,可以用标号来表示跳转的目的地,编译器在编译的

时候会把它替换成地址，标号既可以定义在目的指令同一行的头部，也可以在目的指令前一行单独用一行定义，标号定义的格式是：

标号名:	目的指令	;方法 1
或		
标号名::	目的指令	;方法 2

常用的方法是使用方法 1（标号后跟一个冒号），这时标号的作用域是当前的子程序，在单个子程序中的标号不能同名，否则编译器不知该用哪个地址，但在不同的子程序中可以有相同名称的标号，这意味着不能从一个子程序中用跳转指令跳到另一个子程序中。

需要从一个子程序中用跳转指令跳到另一个子程序中的标号时，可以用方法 2（标号后跟两个冒号）来定义，这时标号的作用域是整个程序，对任何其他子程序都是可见的。

在低版本的 MASM 中，标号在整个程序中是惟一的，子程序中的标号也可以从整个程序的任何地方转入（相当于任何标号都是用方法 2 定义的）。而 Win32 汇编使用的高版本 MASM 中默认的标号作用域是当前的子程序，这是因为为了提供对局部变量和参数的支持，编译器会在子程序的入口处自动加上对堆栈的初始化指令，从子程序的中间进入相当于跳过了这些初始化指令，访问局部变量和参数时会出问题。

当然，如果程序员确认在子程序的中间进入不会引起问题（如用不到局部变量或参数），那完全可以用方法 2 来定义作为入口的标号。

2. MASM 中的@@

在 DOS 时代，为标号起名是个麻烦的事情，因为汇编指令用到跳转指令特别多，任何比较和测试等都要涉及跳转，所以在程序中会有很多标号，在整个程序范围内起个不重名的标号要费一番工夫，结果常常用 addr1 和 addr2 之类的标号一直延续下去，如果后来要在中间插一个标号，那么就常常出现 addr1_1 和 loop10_5 之类奇怪的标号。

实际上，很多标号只会使用一到两次，而且不一定非要起个有意义的名称，如汇编程序中下列代码结构很多：

```

mov     cx, 1234h
cmp     flag, 1
jz      loc1
mov     cx, 1000h
loc1:
...
loop loc1

```

loc1 在别的地方就再也用不到了，对于这种情况，高版本的 MASM 用@@标号去代替它：

```

mov     cx, 1234h
cmp     flag, 1
jz      @F
mov     cx, 1000h
@@:
...
loop @B

```

当用@@做标号时，可以用@F 和@B 来引用它，@F 表示本条指令后的第一个@@标号，@B 表示本条指令前的第一个@@标号，程序中可以有多个@@标号，但@B 和@F 只寻找匹配最近的一个。



不要在间隔太远的代码中使用@@标号，因为在以后的修改中，@@和@B，@F 中间可能会被无意中插入一个新的@@，这样一来，@B 或@F 就会引用到错误的地方去，源程序中@@标号和跳转指令之间的距离最好限制在编辑器能够显示的同一屏幕的范围内。

3.3.2 全局变量

1. 全局变量的定义

全局变量的作用域是整个程序，Win32 汇编的全局变量定义在 .data 或 .data?段内，可以同时定义变量的类型和长度，格式是：

变量名	类型	初始值 1, 初始值 2, ……
变量名	类型	重复数量 dup (初始值 1, 初始值 2, ……)

MASM 中可以定义的变量类型相当多，具体如表 3.2 所示。

表 3.2 变量的类型

名 称	表示方式	缩 写	长度（字节）
字节	Byte	db	1
字	word	dw	2
双字（doubleword）	dword	dd	4
三字（farword）	fword	df	6
四字（quadword）	qword	dq	8
十字节 BCD 码（tenbyte）	tbyte	dt	10
有符号字节（signbyte）	sbyte		1
有符号字（signword）	sword		2
有符号双字（signdword）	sdword		4
单精度浮点数	Real4		4
双精度浮点数	Real8		8
10 字节浮点数	Real10		10

所有使用到变量类型的情况中，只有定义全局变量的时候类型才可以用缩写，现在先来看全局变量定义的几个例子：

.data			
wHour	dw	?	;例 1
wMinute	dw	10	;例 2
_hWnd	dd	?	;例 3
word_Buffer	dw	100 dup (1,2)	;例 4
szBuffer	byte	1024 dup (?)	;例 5
szText	db	'Hello, world!'	;例 6



- 例 1 定义了一个未初始化的 word 类型变量，名称为 wHour。
- 例 2 定义了一个名为 wMinute 的 word 类型变量，其值等于 10。
- 例 3 定义了一个双字类型的变量_hWnd。
- 例 4 定义了一组字，以 0001, 0002, 0001, 0002, ...的顺序在内存中重复 100 遍，一共是 200 个字。
- 例 5 定义了一个 1 024 字节的缓冲区。
- 例 6 定义了一个字符串，总共占用了 12 字节。两头的单引号是定界的符号，并不属于字符串中真正的内容。

在 byte 类型变量的定义中，可以用引号定义字符串和数值定义的方法混用，假设要定义两个字符串“Hello,World!”和“Hello again”，每个字符串后面跟回车和换行符，最后以一个 0 字符结尾，可以定义如下：

szText	db	'Hello,world!','0dh,0ah','Hello again','0dh,0ah,0
--------	----	---

2. 全局变量的初始化值

全局变量在定义中既可以指定初值，也可以只用问号预留空间，在 .data?段中，只能用问号预留空间，因为 .data?不能指定初始值，这里就有一个问题：既然可以用问号预留空间，那么在实际运行的时候，这个未初始化的值是随机的还是确定的呢？答案是 0，所以用问号指定的全局变量如果要以 0 为初始值的话，在程序中可以不特地为它赋值。

3.3.3 局部变量

局部变量这个名称最早源于高级语言，主要是为了定义一些仅在单个函数里面有用的变量而提出的，使用局部变量能带来一些额外的好处，它使程序的模块化封装变得可能，试想一下，如果要用到的变量必须定义在程序的数据段里面，假设在一个子程序中要用到一些变量，当把这个子程序移植到别的程序时，除了把代码移过去以外，还必须把变量定义移过去。而即使把变量定义移过去了，由于这些变量定义在大家都可以用的数据段中，就无法对别的代码保持透明，别的代码有可能有意无意地修改它们。还有，在一个大的工程项目中，存在很多的子程序，所有的子程序要用到的变量全部定义到数据段中，会使数据段变得很大，混在一起的变量也使维护变得非常不方便。

局部变量这个概念出现以后，两个以上子程序都要用到的数据才被定义为全局变量统一放在数据段中，仅在子程序内部使用的变量则放在堆栈中，这样子程序可以编成“黑匣子”的模样，使程序的模块结构更加分明。

局部变量的作用域是单个子程序，在进入子程序的时候，通过修改堆栈指针 esp 来预留出需要的空间，在用 ret 指令返回主程序之前，同样通过恢复 esp 丢弃这些空间，这些变量就随之无效了。它的缺点就是因为空间是临时分配的，所以无法定义含有初始化值的变量，对局部变量的初始化一般在子程序中由指令完成。

在 DOS 时代，低版本的宏汇编本来无所谓全局变量和局部变量，所有的变量都是定义在数

据段里面的，能被所有的子程序或主程序存取，就相当于现在所说的全局变量，用汇编语言在堆栈中定义局部变量是很麻烦的一件事情。要和高级语言做混合编程的时候，程序员往往很痛苦地在边上准备一张表，表上的内容是局部变量名和 ebp 指针的位置关系。

1. 局部变量的定义

MASM 用 local 伪指令提供了对局部变量的支持。定义的格式是：

```
local    变量名 1[[重复数量]][[:类型], 变量名 2[[重复数量]][[:类型]]……
```

local 伪指令必须紧接在子程序定义的伪指令 proc 后、其他指令开始前，这是因为局部变量的数目必须在子程序开始的时候就确定下来，在一个 local 语句定义不下的时候，可以有多个 local 语句，语法中的数据类型不能用表 3.2 中的缩写，如果要定义数据结构，可以用数据结构的名称当做类型。Win32 汇编默认的类型是 dword，如果定义 dword 类型的局部变量，则类型可以省略。当定义数组的时候，可以 [] 括号括起来，不能使用定义全局变量的 dup 伪指令。局部变量不能和已定义的全局变量同名。局部变量的作用域是当前的子程序，所以在不同的子程序中可以有同名的局部变量。

这里有几个定义局部变量的例子：

```
local    loc1[1024]:byte    ;例 1
local    loc2                ;例 2
local    loc3:WNDCLASS      ;例 3
```

- 例 1 定义了一个 1 024 字节长的局部变量 loc1。
- 例 2 定义了一个名为 loc2 的局部变量，类型是默认值 dword。
- 例 3 定义了一个 WNDCLASS 数据结构，名为 loc3。

下面是局部变量使用的一个典型的例子：

```
TestProc proc
        local    @loc1:dword,@loc2:word
        local    @loc3:byte

        mov     eax,@loc1
        mov     ax,@loc2
        mov     al,@loc3
        ret

TestProc endp
```

这是一个名为 TestProc 的子程序，用 local 语句定义了 3 个变量，@loc1 是 dword 类型，@loc2 是 word 类型，@loc3 是 byte 类型，在程序中分别有 3 句存取 3 个局部变量的指令，然后就返回了，编译成可执行文件后，再把它反汇编就得到了以下指令：

```
:00401000 55                push ebp
:00401001 8BEC            mov ebp, esp
:00401003 83C4F8          add esp, FFFFFFF8
:00401006 8B45FC          mov eax, dword ptr [ebp-04]
:00401009 66B45FA         mov ax, word ptr [ebp-06]
```

```

:0040100D 8A45F9      mov al, byte ptr [ebp-07]
:00401010 C9          leave
:00401011 C3          ret

```

可以看到，反汇编后的指令比源程序多了前后两段指令，它们是：

```

:00401000 55          push ebp
:00401001 8BEC        mov ebp, esp
:00401003 83C4F8      add esp, FFFFFFF8
...
:00401010 C9          leave

```

这些就是使用局部变量所必需的指令，分别用于局部变量的准备工作和扫尾工作。执行了 call 指令后，CPU 把返回的地址压入堆栈，再转移到子程序执行，esp 在程序的执行过程中可能随时用到，不可能用 esp 做指针来存取局部变量。大家一定有印象，在介绍寄存器的时候提到过 ebp 寄存器也是以堆栈段为默认数据段的，所以可以用 ebp 做指针，于是，在初始化前，先用一句 push ebp 指令把原来的 ebp 保存起来，然后把 esp 的值放到 ebp 中，供存取局部变量做指针用，再后面就是在堆栈中预留空间了，由于堆栈是向下增长的，所以要在 esp 中加一个负值，FFFFFFF8 就是 -8。慢着！一个 dword 加一个 word 再加一个字节不是 7 吗，为什么是 8 呢？这是因为在 80386 处理器中，以 dword 为界对齐时存取内存速度最快，所以 MASM 宁可浪费一个字节。执行了这 3 句指令后，初始化完成，就可以进行正常的操作了，从指令中可以看出局部变量在堆栈中的位置排列，如表 3.3 所示。

表 3.3 上例中局部变量排列的顺序

ebp 偏移	内 容
ebp+4	由 call 指令推入的返回地址
ebp	push ebp 指令推入的原 ebp 值，然后新的 ebp=现在的 esp
ebp-4	第一个局部变量
ebp-6	第二个局部变量
ebp-7	第三个局部变量

在程序退出的时候，必须把正确的 esp 设置回去，否则，ret 指令会从堆栈中取出错误的地址返回，看程序可以发现，ebp 就是正确的初始 esp 值，因为子程序开始的时候已经有一句 mov ebp, esp，所以要返回的时候只要先 mov esp, ebp，然后再 pop ebp，堆栈就是正确的了。

在 80386 指令集中有一条指令可以在一句中实现 mov esp, ebp 和 pop ebp 的功能，就是 leave 指令，所以，编译器在 ret 指令之前只使用了一句 leave 指令。

明白了局部变量使用的原理，就很容易理解使用时的注意点：ebp 寄存器是关键，它起到保存原始 esp 的作用，并随时用做存取局部变量的指针基址，所以在任何时刻，不要尝试把 ebp 用于别的用途，否则会带来意想不到的后果。



Win32 汇编中局部变量的使用方法可以解释一个很有趣的现象：在 DOS 汇编的时候，如果在子程序中的 push 指令和 pop 指令不配对，那么返回的时候 ret 指令从堆栈里得到的肯定是错误的返回地址，程序也就死掉了。但在 Win32 汇编中，push 指令和 pop 指令不配对可能在逻辑上产生错误，却不会影响子程序正常返回，原因就是在返回的时候 esp 不是靠相同数量的 push 和 pop 指令来保持一致的，而是靠 leave 指令从保存在 ebp 中的原始值中取回来的，也就是说，即使把 esp 改得一塌糊涂也不会影响到子程序的返回，当然，“窍门”就在 ebp，把 ebp 改掉，程序就玩完了！

2. 局部变量的初始化值

显然，局部变量是无法在定义的时候指定初始化值的，因为 local 伪指令只是简单地把空间给留出来，那么开始使用时它里面是什么值呢？和全局变量不一样，局部变量的起始值是随机的，是其他子程序执行后在堆栈里留下的垃圾，所以，对局部变量的值一定要初始化，特别是定义为结构后当参数传递给 API 函数的时候。



在 API 函数使用的大量数据结构中，往往用 0 做默认值，如果用局部变量定义数据结构，初始化时只定义了其中的部分字段，那么剩余字段的当前值可能是编程者预想不到的数值，传给 API 函数后，执行的结果可能是意想不到的，这是初学者很容易忽略的一个问题。所以最好的办法是：在赋值前首先将整个数据结构填 0，然后再初始化要用的字段，这样其余的字段就不必一个个地去填 0 了，RtlZeroMemory 这个 API 函数就是实现填 0 的功能的。

3.3.4 数据结构

数据结构实际上是由多个字段组成的数据“样板”，相当于一种自定义的数据类型，数据结构中间的每一个字段可以是字节、字、双字、字符串或所有可能的数据类型。比如在 API 函数 RegisterClass 中要使用到一个叫做 WNDCLASS 的数据结构，Microsoft 的手册中是如下定义的：

```
typedef struct _WNDCLASS {
    UINT          style;
    WNDPROC       lpfnWndProc;
    int           cbClsExtra;
    int           cbWndExtra;
    HINSTANCE     hInstance;
    HICON         hIcon;
    HCURSOR       hCursor;
    HBRUSH        hbrBackground;
    LPCTSTR       lpszMenuName;
    LPCTSTR       lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

注意，这是 C 语言格式的，这个数据结构包含了 10 个字段，字段的名称是 style，lpfnWndProc 和 cbClsExtra 等，前面的 UINT 和 WNDPROC 等是这些字段的类型，在汇编中，数

据结构的写法如下：

结构名	struct	
字段 1	类型	?
字段 2	类型	?
.....		
结构名	ends	

上面的 WNDCLASS 结构定义用汇编的格式来表示就是：

WNDCLASS	struct	
style	DWORD	?
lpfnWndProc	DWORD	?
cbClsExtra	DWORD	?
cbWndExtra	DWORD	?
hInstance	DWORD	?
hIcon	DWORD	?
hCursor	DWORD	?
hbrBackground	DWORD	?
lpszMenuName	DWORD	?
lpszClassName	DWORD	?
WNDCLASS	ends	

和大部分的常量一样，几乎所有 API 所涉及的数据结构在 Windows.inc 文件中都已经有定义了。要注意的是，定义了数据结构实际上只是定义了一个“样板”，上面的定义语句并不会在哪个段中产生数据，与 Word 中使用各种“信纸”与“文书”等模板类似，定义了数据结构以后就可以多次在源程序中用这个“样板”当做数据类型来定义数据，使用数据结构在数据段中定义数据的方法如下：

	.data?	
stWndClass	WNDCLASS	<>
	

或者：

	.data	
stWndClass	WNDCLASS	<1, 1, 1, 1, 1, 1, 1, 1, 1, 1>
	

这个例子定义了一个以 WNDCLASS 为结构的变量 stWndClass，第一段的定义方法是未初始化的定义方法，第二段是在定义的同时指定结构中各字段的初始值，各字段的初始值用逗号隔开，在这个例子中 10 个字段的初始值都指定为 1。

在汇编中，数据结构的引用方法有好几种，以上面的定义为例，如果要使用 stWndClass 中的 lpfnWndProc 字段，最直接的办法是：

mov	eax, stWndClass.lpfnWndProc
-----	-----------------------------

它表示把 lpfnWndProc 字段的值放入 eax 中去，假设 stWndClass 在内存中的地址是

403000h, 这句指令会被编译成 `mov eax, [403004h]`, 因为 `lpfnWndProc` 是 `stWndClass` 中的第二个字段, 第一个字段是 `dword`, 已经占用了 4 字节的空间。

在实际使用中, 常常有使用指针存取数据结构的情况, 如果使用 `esi` 寄存器做指针寻址, 可以使用下列语句完成同样的功能:

```
mov     esi, offset stWndClass
mov     eax, [esi + WNDCLASS.lpfndProc]
```

注意: 第二句是 `[esi + WNDCLASS.lpfndProc]` 而不是 `[esi + stWndClass.lpfndProc]`, 因为前者会被编译成 `mov eax, [esi+4]`, 而后者会被编译成 `mov eax, [esi+403004h]`, 后者的结果显然是错误的! 如果要对一个数据结构中的大量字段进行操作, 这种写法显然比较烦琐, MASM 还有一个用法, 可以用 `assume` 伪指令把寄存器预先定义为结构指针, 再进行操作:

```
mov     esi, offset stWndClass
assume  esi:ptr WNDCLASS
mov     eax, [esi].lpfnWndProc
...
assume  esi:nothing
```

这样, 使用寄存器也可以用逗号引用字段名, 程序的可读性比较好。这样的写法在最后编译成可执行程序的时候产生同样的代码。注意: 在不再使用 `esi` 寄存器做指针的时候要用 `assume esi:nothing` 取消定义。

结构的定义也可以嵌套, 如果要定义一个新的 `NEW_WNDCLASS` 结构, 里面包含一个老的 `WNDCLASS` 结构和一个新的 `dwOption` 字段, 那么可以如下定义:

```
NEW_WNDCLASS struct

dwOption d          word      ?
oldWndClass          WNDCLASS  <>

NEW_WNDCLASS ends
```

假设现在 `esi` 是指向一个 `NEW_WNDCLASS` 的指针, 那么引用里面嵌套的 `oldWndClass` 中的 `lpfnWndProc` 字段时, 就可以用下面的语句:

```
mov     eax, [esi].oldWndClass.lpfndProc
```

结构的嵌套在 Windows 的数据定义中也是常有的, 比如在第 13 章 13.3 节中使用的 `DEBUG_EVENT` 结构中竟然使用了 4 层数据结构的嵌套。熟练掌握数据结构的使用对 Win32 汇编编程是很重要的!

3.3.5 变量的使用

1. 以不同的类型访问变量

这个话题有点像 C 语言中的数据类型强制转换, C 语言中的类型转换指的是把一个变量的内容转换成另外一种类型, 转换过程中, 数据的内容已经发生了变化, 如把浮点数转换成整数后,

小数点后的内容就丢失了。在 MASM 中以不同的类型访问不会对变量造成影响。

举一个简单的例子，先以 db 方式定义一个缓冲区：

```
szBuffer    db    1024 dup (?)
```

然后从其他地方取得了数据，但数据的格式是以字方式组织的，要处理数据，最有效的方法是两个字节两个字节地处理，但如果在程序中把 szBuffer 的值放入 ax：

```
mov    ax,szBuffer
```

编译器会报一个错：

```
error A2070: invalid instruction operands
```

意思是无效的指令操作，为什么呢？因为 szBuffer 是用 db 定义的，而 ax 的尺寸是一个 word，等于两个字节，尺寸不符合。MASM 中，如果要用指定类型之外的长度访问变量，必须显式地指出要访问的长度，这样，编译器忽略语法上的长度检验，仅使用变量的地址。使用的方法是：

类型 ptr 变量名

类型可以是 byte, word, dword, fword, qword, real8 和 real10。如：

```
mov    ax,word ptr szBuffer
mov    eax,dword ptr szBuffer
```

上述语句能通过编译，当然，类型必须和操作的寄存器长度匹配。在这里要注意的是，指定类型的参数访问并不会去检测长度是否溢出，看下面一段代码：

```
.data
bTest1    db    12h
wTest2    dw    1234h
dwTest3    dd    12345678h
...
.code
...
mov    al,bTest1
mov    ax,word ptr bTest1
mov    eax,dword ptr bTest1
...
```

上面的程序片断，每一句执行后寄存器中的值是什么呢，mov al,bTest1 这一句很显然使 al 等于 12h，下面的两句呢，ax 和 eax 难道等于 0012h 和 00000012h 吗？实际运行结果很“奇怪”，竟然是 3412h 和 78123412h，为什么呢？先来看反汇编的内容：

```
; .data 段中的变量
:00403000 12 34 12 78 56 34 12 ...
          |   |   |   |
          |   |   |   └─> dwTest3
          |   |   └─> wTest2
          |   └─> bTest1
```

: .code 段中的代码	
:00401000 A000304000	mov al, byte ptr [00403000]
:00401005 66A100304000	mov ax, word ptr [00403000]
:0040100B A100304000	mov eax, dword ptr [00403000]

.data 段中的变量是按顺序从低地址往高地址排列的, 对于超过一个字节的数, 80386 处理器的数据排列方式是低位数据在低地址, 所以 wTest2 的 1234h 在内存中的排列是 34h 12h, 因为 34h 是低位。同样, dwTest3 在内存中以 78h 56h 34h 12h 从低地址往高地址存放, 在执行指令 mov ax, word ptr bTest1 的时候, 是从 bTest1 的地址 403000h 处取一个字, 其长度已经超过了 bTest1 的范围并落到了 wTest2 中, 从内存中看, 是取了 bTest1 的数据 12h 和 wTest2 的低位 34h, 在这两个字节中, 12h 位于低地址, 所以 ax 中的数值是 3412h。同样道理, 看另一条指令:

mov	eax, dword ptr bTest1
-----	-----------------------

这条指令取了 bTest1, wTest2 的全部和 dwTest3 的最低位 78h, 在内存中的排列是 12h 34h 12h 78h, 所以 eax 等于 78123412h。

这个例子说明了汇编中用 ptr 强制覆盖变量长度的时候, 实质上只用了变量的地址, 编译器并不会考虑定界的问题, 程序员在使用的时候必须对内存中的数据排列有个全局概念, 以免越界存取到意料之外的数据。

如果程序员的本意是类似于 C 语言的强制类型转换, 想把 bTest1 的一个字节扩展到一个字或一个双字再放到 ax 或 eax 中, 高位保持 0 而不是越界存取到其他的变量, 可以用 80386 的扩展指令来实现。80386 处理器提供的 movzx 指令可以实现这个功能, 例如:

movzx	ax, bTest1	;例 1
movzx	eax, bTest1	;例 2
movzx	eax, cl	;例 3
movzx	eax, ax	;例 4

- 例 1 把单字节变量 bTest1 的值扩展到 16 位放入 ax 中。
- 例 2 把单字节变量 bTest1 的值扩展到 32 位放入 eax 中。
- 例 3 把 cl 中的 8 位值扩展到 32 位放入 eax 中。
- 例 4 把 ax 中的 16 位值扩展到 32 位放入 eax 中。

用 movzx 指令进行数据长度扩展是 Win32 汇编中经常用到的技巧, 该指令总是将扩展的数据位用 0 代替。使用另一条指令 movsx 可以完成带符号位的扩展, 当被扩展数据的最高位为 0 时, 效果和 movzx 指令相同; 当最高位为 1 时, 则扩展部分的数据位全部用 1 填充。

2. 变量的尺寸和数量

在源程序中用到变量的尺寸和数量的时候, 可以用 sizeof 和 lengthof 伪指令来实现, 格式是:

sizeof	变量名、数据类型或数据结构名
lengthof	变量名、数据类型或数据结构名

sizeof 伪指令可以取得变量、数据类型或数据结构以字节为单位的长度，lengthof 可以取得变量中数据的项数。假如定义了以下数据：

```

stWndClass      WNDCLASS <>
szHello         db  'Hello,world!',0
dwTest         dd  1,2,3,4
...
.code
...
mov     eax, sizeof stWndClass
mov     ebx, sizeof WNDCLASS
mov     ecx, sizeof szHello
mov     edx, sizeof dword
mov     esi, sizeof dwTest

```

执行后 eax 的值是 stWndClass 结构的长度 40，ebx 同样是 40，ecx 的值是 13，就是“Hello,world!”字符串的长度加上一个字节的 0 结束符，edx 的值是一个双字的长度：4，而 esi 则等于 4 个双字的长度 16。

如果把所有的 sizeof 换成 lengthof，那么 eax 会等于 1，因为只定义了 1 项 WNDCLASS，而 ecx 同样等于 13，esi 则等于 4，而 lengthof WNDCLASS 和 lengthof dword 是非法的用法，编译程序会报错。

要注意的是，sizeof 和 lengthof 的数值是编译时候产生的，由编译器直接替换到指令中去，上边的指令最后产生的代码就是：

```

mov     eax, 40
mov     ebx, 40
mov     ecx, 13
mov     edx, 4
mov     esi, 16

```



如果为了把 Hello 和 World 分两行定义，szHello 是这样定义的：

```

szHello      db      'Hello',0dh,0ah
              db      'World',0

```

那么 sizeof szHello 是多少呢？注意！是 7 而不是 13，MASM 中的变量定义只认一行，后一行 db 'World',0 实际上是另一个没有名称的数据定义，编译器认为 sizeof szHello 是第一行字符的数量。虽然把 szHello 的地址当参数传给 MessageBox 等函数显示时会把两行都显示出来，但严格地说这是越界使用变量。虽然在实际的应用中这样定义长字符串的用法很普遍，因为如果要显示一屏幕帮助，一行是不够的，但要注意的是：要用到这种字符串的长度时，千万不要用 sizeof 去表示，最好是在程序中用 strlen 函数去计算。

3. 获取变量地址

获取变量地址的操作对于全局变量和局部变量是不同的。

对于全局变量，它的地址在编译的时候已经由编译器确定了，它的用法大家都不陌生：

```

mov     寄存器, offset 变量名

```

其中 `offset` 是取变量地址的伪操作符，和 `sizeof` 伪操作符一样，它仅把变量的地址代到指令中去，这个操作是在编译时而不是在运行时完成的。

对于局部变量，它是用 `ebp` 来做指针操作的，假设 `ebp` 的值是 `40100h`，那么局部变量 1 的地址是 `ebp-4` 即 `400FCh`，由于 `ebp` 的值随着程序的执行环境不同可能是不同的，所以局部变量的地址值在编译的时候也是不确定的，不可能用 `offset` 伪操作符来获取它的地址。

80386 处理器中有一条指令用来取指针的地址，就是 `lea` 指令，如：

```
lea    eax, [ebp-4]
```

该指令可以在运行时按照 `ebp` 的值实际计算出地址放到 `eax` 中。

如果要在 `invoke` 伪指令的参数中用到一个局部变量的地址，该怎么办呢？参数中是不可能写入 `lea` 指令的，用 `offset` 又是不对的。MASM 对此有一个专用的伪操作符 `addr`，其格式为：

```
addr    局部变量名和全局变量名
```

当 `addr` 后跟全局变量名的时候，编译器自动按照 `offset` 的用法来使用；当 `addr` 后面跟局部变量名的时候，编译器会自动用 `lea` 指令先把地址取到 `eax` 中，然后用 `eax` 来代替变量地址使用。

要注意的是：对局部变量取地址的时候，`addr` 伪操作符只能用在 `invoke` 的参数中，不能用在如下的 `mov` 指令中，这种限制很好理解，因为这种情况下，`lea` 指令如何能被代到语句里面呢：

```
mov    eax, addr 局部变量名    ;注意：这是错误的用法
```



假设在一个子程序中有如下 `invoke` 指令：

```
invoke Test, eax, addr szHello
```

其中 `Test` 是一个需要两个参数的子程序，`szHello` 是一个局部变量，会发生什么结果呢？编译器会把 `invoke` 伪指令和 `addr` 翻译成下面这个模样：

```
lea    eax, [ebp-4]
push   eax        ;参数 2: addr szHello
push   eax        ;参数 1: eax
call   Test
```

发现了什么？到 `push` 第一个参数 `eax` 之前，`eax` 的值已经被 `lea eax, [ebp-4]` 指令覆盖了！也就是说，要用到的 `eax` 的值不再有效，所以，当在 `invoke` 中使用 `addr` 伪操作符时，注意在它的左边不能用 `eax`，否则 `eax` 的值会被覆盖掉，当然 `eax` 用在 `addr` 右边的参数中用是可以的。幸亏 MASM 编译器对这种情况有如下错误提示：

```
error A2133: register value overwritten by INVOKE
```

否则，不知道又会引出多少莫名其妙的错误！

3.4 使用子程序

当程序中相同功能的一段代码用得比较频繁时，可以将它分离出来写成一个子程序，在主

程序中用 `call` 指令来调用它。这样可以不用重复写相同的代码，仅仅用 `call` 指令就可以完成多次同样的工作了。Win32 汇编中的子程序也采用堆栈来传递参数，这样就可以用 `invoke` 伪指令来进行调用和语法检查工作。

3.4.1 子程序的定义

子程序的定义方式如下所示。

```
子程序名  proc [距离][语言类型][可视区域][USES 寄存器列表][, 参数:类型]... [VARARG]
            local 局部变量列表
```

指令

```
子程序名  endp
```

`proc` 和 `endp` 伪指令定义了子程序开始和结束的位置，`proc` 后面跟的参数是子程序的属性和输入参数。子程序的属性有：

- 距离——可以是 NEAR, FAR, NEAR16, NEAR32, FAR16 或 FAR32，Win32 中只有一个平坦的段，无所谓距离，所以对距离的定义往往忽略。
- 语言类型——表示参数的使用方式和堆栈平衡的方式，可以是 StdCall, C, SysCall, BASIC、FORTRAN 和 PASCAL，如果忽略，则使用程序头部 `.model` 定义的值。
- 可视区域——可以是 PRIVATE, PUBLIC 和 EXPORT。PRIVATE 表示子程序只对本模块可见；PUBLIC 表示对所有的模块可见（在最后编译链接完成的 `.exe` 文件中）；EXPORT 表示是导出的函数，当编写 DLL 的时候要将某个函数导出的时候可以这样使用。默认的设置是 PUBLIC。
- USES 寄存器列表——表示由编译器在子程序指令开始前自动安排 `push` 这些寄存器的指令，并且在 `ret` 前自动安排 `pop` 指令，用于保存执行环境，但笔者认为不如自己在开头和结尾用 `pushad` 和 `popad` 指令一次保存和恢复所有寄存器来得方便。
- 参数和类型——参数指参数的名称，在定义参数名的时候不能跟全局变量和子程序中的局部变量重名。对于类型，由于 Win32 中的参数类型只有 32 位（`dword`）一种类型，所以可以省略。在参数定义的最后还可以跟 VARARG，表示在已确定的参数后还可以跟多个数量不确定的参数，在 Win32 汇编中惟一使用 VARARG 的 API 就是 `wsprintf`，类似于 C 语言中的 `printf`，其参数的个数取决于要显示的字符串中指定的变量个数。

完成了定义之后，可以用 `invoke` 伪指令来调用子程序，当 `invoke` 伪指令位于被调用的子程序代码之前的时候，编译器处理到 `invoke` 语句的时候还没有扫描到子程序的定义信息，所以会有以下错误信息：

```
error A2006: undefined symbol : 子程序名
```

这并不是说子程序的编写有错误，而是 `invoke` 伪指令无法得知子程序的定义情况，所以无法进行参数的检测。在这种情况下，为了让 `invoke` 指令能正常使用，必须在程序的头部用 `proto` 伪操作定义子程序的信息，“提前”告诉 `invoke` 语句关于子程序的信息，`proto` 的用法见 3.2.2 节。当然，如果被调用的子程序定义在 `invoke` 语句前面的话，`proto` 语句就可以省略了。

至：

了解了子程序的定义方法后，让我们继续深入了解子程序的使用细节。在调用子程序时，参数的传递是通过堆栈进行的，也就是说，调用者把要传递给子程序的参数压入堆栈，子程序在堆栈中取出相应的值再使用，比如，如果要调用：

SubRouting (Var1, Var2, Var3)

经过编译后的最终代码可能是（注意只是“可能”）：

push	Var3
push	Var2
push	Var1
call	SubRouting
add	esp, 12

	C	SysCall	StdCall	BASIC	FORTRAN	PASCAL
最先入栈参数	右	右	右	左	左	左
清除堆栈者	调用者	子程序	子程序	子程序	子程序	子程序
允许使用 VARARG	是	是	是 ^注	否	否	否

注: VARARG 表示参数的个数可以是不确定的, 如 `wsprintf` 函数, `StdCall` 的堆栈清除平时是由子程序完成的, 但使用 `VARARG` 时是由调用者清除的。

[illegible]

编译后再进行反汇编，看编译器是如何转换处理不同类型的子程序的：

77

可以清楚地看到，在参数入栈顺序上，C 类型和 StdCall 类型是先把右边的参数先压入堆栈，而 PASCAL 类型是先把左边的参数压入堆栈。在堆栈平衡上，C 类型是在调用者在使用 call 指令完成后，自行用 add esp, 8 指令把 8 个字节的参数空间清除，而 PASCAL 和 StdCall 的调用者则不管这个事情，堆栈平衡的事情是由子程序用 ret 8 来实现的（ret 指令后面加一个操作数表示在 ret 后把堆栈指针 esp 加上操作数）。

因为 Win32 约定的类型是 StdCall，所以在程序中调用子程序或系统 API 后，不必自己来平衡堆栈，免去了很多麻烦。

存取参数和局部变量都是通过堆栈来定义的，和存取局部变量类似，参数的存取也是通过 ebp 做指针来完成的。在表 3.3 中，已经对 ebp 指针和局部变量的对应关系做了分析，现在来分析一下 ebp 指针和参数之间的对应关系，注意，这里是以 Win32 中的 StdCall 为例，不同的语言类型，指针的顺序可能是不同的。

假定在一个子程序中有两个参数，主程序调用时在 push 第一个参数前的堆栈指针 esp 为 X，那么压入两个参数后的 esp 为 X-8，程序开始执行 call 指令，call 指令把返回地址压入堆栈，这时候 esp 为 X-C，接下去是子程序中用 push ebp 来保存 ebp 的值，esp 变为 X-10，再执行一句 mov ebp, esp，就可以开始用 ebp 存取参数和局部变量了，图 3.4 说明了这个过程。

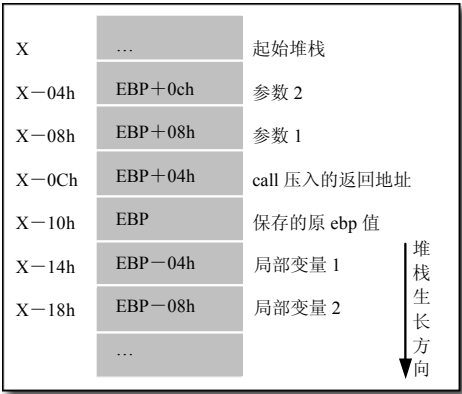


图 3.4 ebp 指针、参数和局部变量的关系

在源程序中，由于参数、局部变量和 ebp 的关系是由编译器自动维护的，所以读者不必关心它们的具体关系，但到了用 Soft-ICE 等工具来分析其他软件的时候，遇到调用子程序的时候一定要先看清楚它们之间的类型差别。

在子程序中使用参数，可以使用与存取局部变量同样的方法，因为这两者的构造原理几乎一模一样，所以，在子程序中有 invoke 语句时，如果要用到输入参数的地址当做 invoke 的参数，同样要遵循局部变量的使用方式，不能用 offset 伪操作符，只能用 addr 来完成。同样，所有对局部变量使用的限制几乎都可以适用于参数。

3.5 高级语法

以前高级语言和汇编的最大差别就是条件测试、分支和循环等高级语法。高级语言中，程序员可以方便地用类似于 if, case, loop 和 while 等语句来构成程序的结构流程，不仅条理清楚、一目了然，而且维护性相当好。而汇编程序员呢？只能在 cmp 指令后面绞尽脑汁地想究竟用几十种跳转语句中的哪一种，这里就能列出近三十个条件跳转指令来：ja, jae, jb, jbe, jc, je, jg, jge, jl, jle, jna, jnae, jnb, jnbe, jnc, jne, jng, jnge, jnl, jnle, jno, jnp, jns, jnz, jo, jp, jpe, jpo 和 jz 等。虽然其中的很多指令我们一辈子也不会用到，但就是这些指令和一些 loop, loopnz, 以及被 loop 涉及的 ecx 等寄存器纠缠在一起，使在汇编中书写结构清晰、可读性好的代码变得相当困难，这也是很多人视汇编为畏途的一个原因。

现在好了，MASM 中新引入了一系列的伪指令，涉及条件测试、分支和循环语句，利用它们，汇编语言有了与高级语言一样的结构，配合对局部变量和调用参数等高级语言中常见元素的支持，为使用 Win32 汇编编写大规模的应用程序奠定了基础。

3.5.1 条件测试语句

在高级语言中，所有的分支和循环语句首先要涉及条件测试，也就是涉及一个表达式的结果是“真”还是“假”的问题，表达式中往往有用来做比较和计算的操作符，MASM 也不例外，这就是条件测试语句。

MASM 条件测试的基本表达式是：

寄存器或变量 操作符 操作数

两个以上的表达式可以用逻辑运算符连接：

(表达式 1) 逻辑运算符 (表达式 2) 逻辑运算符 (表达式 3) ...

允许的操作符和逻辑运算符如表 3.5 所示。

表 3.5 条件测试中的操作符

操作符和逻辑运算	操 作	用 途
==	等于	变量和操作数之间的比较
!=	不等于	变量和操作数之间的比较
>	大于	变量和操作数之间的比较
>=	大于等于	变量和操作数之间的比较
<	小于	变量和操作数之间的比较
<=	小于等于	变量和操作数之间的比较
&	位测试	将变量和操作数做“与”操作
!	逻辑取反	对变量取反或对表达式的结果取反
&&	逻辑与	对两个表达式的结果进行逻辑“与”操作
	逻辑或	对两个表达式的结果进行逻辑“或”操作

举例如下，左边为表达式，右边是表达式为“真”的条件：

<code>x==3</code>	;x 等于 3
<code>eax!=3</code>	;eax 不等于 3
<code>(y>=3)&&ebx</code>	;y 大于等于 3 且 ebx 为非零值
<code>(z&1) !eax</code>	;z 和 1 进行“与”操作后非零或 eax 取反后非零
	;也就是说 z 的位 0 等于 1 或者 eax 为零

细心的读者一定会发现，MASM 的条件测试采用的是和 C 语言相同的语法。如 ! 和 & 是对变量的操作符（取反和“与”操作），|| 和 && 是表达式结果之间的逻辑“与”和逻辑“或”，而 ==、!=、>、< 等是比较符。同样，对于不含比较符的单个变量或寄存器，MASM 也是将所有非零值认为是“真”，零值认为是“假”。

MASM 的条件测试语句有几个限制，首先是表达式的左边只能是变量或寄存器，不能为常数；其次表达式的两边不能同时为变量，但可以同时是寄存器。这些限制来自于 80x86 的指令，因为条件测试伪操作符只是简单地把每个表达式翻译成 `cmp` 或 `test` 指令，80x86 的指令集中没有 `cmp 0, eax` 之类的指令，同时也不允许直接操作两个内存中的数，所以对这两个限制是很好理解的。

除了这些和高级语言类似的条件测试伪操作，汇编语言还有特殊的要求，就是程序中常常要根据系统标志寄存器中的各种标志位来做条件跳转，这些在高级语言中是用不到的，所以又增加了以下一些标志位的状态指示，它们本身相当于一个表达式：

<code>CARRY?</code>	表示 Carry 位是否置位
<code>OVERFLOW?</code>	表示 Overflow 位是否置位
<code>PARITY?</code>	表示 Parity 位是否置位
<code>SIGN?</code>	表示 Sign 位是否置位
<code>ZERO?</code>	表示 Zero 位是否置位

要测试 `eax` 等于 `ebx` 同时 Zero 位置位，条件表达式可以写为：

```
(eax==ebx) && ZERO?
```

要测试 `eax` 等于 `ebx` 同时 Zero 位清零，条件表达式可以写为：

```
(eax==ebx) && ! ZERO?
```

与 C 语言的条件测试相同，MASM 的条件测试伪指令并不会改变被测试的变量或寄存器的值，只是进行“测试”而已，到最后它会被编译器翻译成类似于 `cmp` 或 `test` 之类的比较或位测试指令。

3.5.2 分支语句

分支语句用来根据条件表达式测试的真假执行不同的代码模块，MASM 中的分支语句的语法如下：

```
.if 条件表达式 1
    表达式 1 为“真”时执行的指令
[.elseif 条件表达式 2]
    表达式 2 为“真”时执行的指令
```



```

[.elseif 条件表达式 3]
    表达式 3 为“真”时执行的指令
...
[.else]
    所有表达式为“否”时执行的指令
.endif

```

注意：关键字 if/elseif/else/endif 的前面有个小数点，如果不加小数点，就变成宏汇编中的条件汇编伪操作了，结果可是天差地别。

为了说明编译器究竟是如何处理这些伪指令的，先写一段如下的源代码：

```

.if eax && (ebx >= dwX) || !(dwY != ecx)
    mov     esi,1
.elseif edx
    mov     esi,2
.elseif esi & 1
    mov     esi,3
.elseif ZERO? && CARRY?
    mov     esi,4
.endif

```

然后反汇编，得到了以下的汇编指令：

```

;      .if eax
:00401000 0BC0                      or eax, eax
:00401002 7408                      je 0040100C
;      (ebx >= dwX)
:00401004 3B1D00304000             cmp ebx, dword ptr [00403000]
:0040100A 7308                      jnb 00401014
;      (dwY != ecx)
:0040100C 390D04304000             cmp dword ptr [00403004], ecx
:00401012 7507                      jne 0040101B
:00401014 BE01000000           mov esi, 00000001
:00401019 EB23                      jmp 0040103E
;      elseif edx
:0040101B 0BD2                      or edx, edx
:0040101D 7407                      je 00401026
:0040101F BE02000000           mov esi, 00000002
:00401024 EB18                      jmp 0040103E
;      elseif esi & 1
:00401026 F7C601000000           test esi, 00000001
:0040102C 7407                      je 00401035
:0040102E BE03000000           mov esi, 00000003
:00401033 EB09                      jmp 0040103E
;      ZERO?
:00401035 7507                      jne 0040103E
;      CARRY?
:00401037 7305                      jnb 0040103E
:00401039 BE04000000           mov esi, 00000004
:0040103E ...

```

可以看到，MASM 编译器对这些条件分支伪指令优化得相当好，看到这些反汇编后的指令，

惟一的感觉是好像又回到了 DOS 汇编时代分支指令堆中，从这里可以发现，这些伪指令把汇编源程序的可读性基本上提高到了高级语言的水平。

分析反汇编代码可以发现，在不同的条件满足之后，先是执行满足条件后需要执行的指令，如上面的 `mov esi, 0001` 和 `mov esi, 0002` 等指令，这些指令执行后，后面都有一句直接跳转的指令 `jmp 0040103E`，`0040103E` 地址对应整个条件分支结构的尾部，这意味着，由 `.if/.elseif/.else/.endif` 条件分支伪指令构成的分支结构只能有一个条件被满足，也就是说，程序按照从上到下的各个条件表达式，顺序判断，当第一个条件表达式满足的时候，执行相应的代码，然后就忽略掉下面所有的其他条件表达式，即使后面有另一个满足条件时也是如此！

如果需要构成的分支结构对于所有的表达式为“真”都要执行相应的代码，可以利用多个 `.if/.endif` 来完成，如下所示：

```
.if      表达式 1
        表达式 1 为“真”要执行的指令
.endif
.if      表达式 2
        表达式 2 为“真”要执行的指令
.endif
...
```



使用 `.if/.else/.endif` 构成分支伪指令的时候，不要漏写前面的小数点，`if/else/endif` 是宏汇编中条件汇编宏操作的伪操作指令，作用是根据条件决定在最后的可执行文件中包不包括某一段代码。比如在程序的调试阶段：

```
DEBUG      equ 1
...
if         DEBUG
        invoke MessageBox, 0, offset szText, offset szCaption, MB_OK
endif
```

该代码用来显示一个调试信息，当程序正式发行时，将第一句改为 `DEBUG equ 0`，然后再编译，那么可执行文件中根本不会包括这段代码，这和 `.if/.else/.endif` 构成分支的伪指令完全是两回事。

3.5.3 循环语句

循环是重复执行的一组指令，MASM 的循环伪指令可以根据条件表达式的真假来控制循环是否继续，也可以在循环体中直接退出，使用循环的语法是：

```
.while  条件测试表达式
    指令
    [.break [.if 退出条件]]
    [.continue]
.endw
```

或

```
.repeat
```

```

指令
[.break [.if 退出条件]]
[.continue]
.until 条件测试表达式 (或.untilcxz [条件测试表达式])

```

.while/.endw 循环首先判断条件测试表达式，如果结果是“真”，则执行循环体内的指令，结束后再回到 .while 处判断表达式，如此往复，一直到表达式结果为“假”为止。.while/.endw 指令有可能一遍也不会执行到循环体内的指令，因为如果第一次判断表达式时就遇到结果为“假”的情况，那么就直接退出循环。

.repeat/.until 循环首先执行一遍循环体内的指令，然后再判断条件测试表达式，如果结果为“真”的话，就退出循环，如果为“假”，则返回 .repeat 处继续循环，可以看出，.repeat/.until 不管表达式的值如何，至少会执行一遍循环体内的指令。

也可以把条件表达式直接设置为固定值，这样就可以构建一个无限循环，对于 .while/.end 直接使用 TRUE，对 .repeat/.until 直接使用 FALSE 来当表达式就是如此，这种情况下，可以使用 .break 伪指令强制退出循环，如果 .break 伪指令后面跟一个 .if 测试伪指令的话，那么当退出条件为“真”时才执行 .break 伪指令。

在循环体中也可以用 .continue 伪指令忽略以后的指令，遇到 .continue 伪指令时，不管下面还有没有其他循环体中的指令，都会直接回到循环头部开始执行。

同样，为了深入了解 MASM 编译器把循环伪指令变成了什么，下面对比一段源程序和反汇编后的代码。首先是源程序：

```

.while eax > 1
    mov     esi,1
    .break  .if ebx
    .continue
    mov     esi,2
.endw
.repeat
    mov     esi,1
    .break  .if !ebx
    .continue
    mov     esi,2
.until eax > 1
.repeat
    mov     esi,1
    .break
.untilcxz

```

以下是反汇编后的代码：

```

;          .while    第一个循环开始
:00401000 EB10                jmp 00401012
:00401002 BE01000000          mov esi, 00000001
:00401007 0BDB                or ebx, ebx
;          .break .if ebx
:00401009 750C                jne 00401017
;          .continue

```

```

:0040100B EB05                jmp 00401012
:0040100D BE02000000          mov esi, 00000002
;    .while eax > 1
:00401012 83F801                cmp eax, 00000001
:00401015 77EB                ja 00401002
;    .repeat 第二个循环开始
:00401017 BE01000000          mov esi, 00000001
;    .break .if !ebx
:0040101C 0BDB                or ebx, ebx
:0040101E 740C                je 0040102C
;    .continue
:00401020 EB05                jmp 00401027
:00401022 BE02000000          mov esi, 00000002
;    .until eax > 1
:00401027 83F801                cmp eax, 00000001
:0040102A 76EB                jbe 00401017
;    .repeat 第三个循环开始
:0040102C BE01000000          mov esi, 00000001
;    .break
:00401031 EB02                jmp 00401035
;    .untilcxz
:00401033 E2F7                loop 0040102C ;注意这里是 loop 指令!

```

对比伪指令和翻译成的实际指令，可以对循环的伪指令有更好的理解：`.break` 翻译成一个跳转指令跳到循环结束的地方，`.continue` 是一个无条件跳转指令跳到循环开始的地方，`.while` 是先比较条件再执行循环体，而 `.repeat` 是先执行循环体再比较条件的。

对指令的分析中可以发现，`.while/.endw` 和 `.repeat/.until` 循环没有使用 `loop` 指令的优势，因为 `loop` 指令可以自动递减 `ecx` 的值来控制循环，不使用 `loop` 将会在循环体内多设置一条参数递减的指令，但不用 `loop` 指令的好处是带来更多的灵活性。为了弥补这个缺陷，可以使用 `.repeat/.untilcxz` 伪指令，编译器将会强制使用 `loop` 指令来完成循环，当然，在这种用法中，程序员必须在循环开始前正确设置 `ecx` 的值。

如果又想用 `loop` 指令来构成循环又要使用条件表达式怎么办，这时同样可以在 `.untilcxz` 伪指令后加条件测试语句，只不过这时候有很大的限制，第一只能是单个条件表达式，不能用 `&&` 或 `||` 来构成多项表达式了；第二即使是单个表达式中，也只能用 `==` 或 `!=` 操作符，不能用其他比较大小的操作符，因为这时编译器的翻译方式是在一个比较指令后使用 `loopz` 或 `loopnz` 来构成循环，这个指令不能测试其他标志位。



在分支和循环的伪指令反汇编后可以发现，在使用 `>`、`>=`、`<` 和 `<=` 比较符时，MASM 的伪指令总是将比较以后的跳转指令使用为 `jb` 和 `jnb` 等无符号数比较跳转的指令，这就意味着，MASM 的条件测试总是把操作数当做无符号数看待，这样，假设 `eax` 等于 1，那么表达式 `(eax > -1)` 的值是“假”，因为 -1 表示为 `0xffffffffh`，如果当做无符号数看，它反而是最大的数！如果程序中需要构造有符号数的比较分支或循环结构，那么必须另外用 `j1` 和 `jg` 等有符号数比较跳转的指令来完成，使用条件测试配合分支或循环伪指令可能会得到错误的结果！

3.6 代码风格

随着程序功能的增加和版本的提高，程序越来越复杂，源文件也越来越多，风格规范的源程序会对软件的升级、修改和维护带来极大的方便，要想开发一个成熟的软件产品，必须在编写源程序的时候就有条不紊、细致严谨。

在编程中，在程序排版、注释、命名和可读性等问题上都有一定的规范，虽然编写可读性良好的代码并不是必然的要求（世界上还有难懂代码比赛，看谁的代码最不好读懂！），但好的代码风格实际上是为自己将来维护和使用这些代码节省时间。本节就是对汇编语言代码风格的建议。

3.6.1 变量和函数的命名

1. 匈牙利表示法

匈牙利表示法主要用在变量和子程序的命名，这是现在大部分程序员都在使用的命名约定。“匈牙利表示法”这个奇怪的名字是为了纪念匈牙利籍的 Microsoft 程序员 Charles Simonyi，他首先使用了这种命名方法。

匈牙利表示法用连在一起的几个部分来命名一个变量，格式是类型前缀加上变量说明，类型用小写字母表示，如用 h 表示句柄，用 dw 表示 double word，用 sz 表示以 0 结尾的字符串等，说明则用首字母大写的几个英文单词组成，如 TimeCounter，NextPoint 等，可以令人一眼看出变量的含义来，在汇编语言中常用的类型前缀有：

b	表示 byte
w	表示 word
dw	表示 dword
h	表示句柄
lp	表示指针
sz	表示以 0 结尾的字符串
lpsz	表示指向 0 结尾的字符串的指针
f	表示浮点数
st	表示一个数据结构

这样一来，变量的意思就很好理解：

hWinMain	主窗口的句柄
dwTimeCount	时间计数器，以双字定义
szWelcome	欢迎信息字符串，以 0 结尾
lpBuffer	指向缓冲区的指针
stWndClass	WNDCLASS 结构
...	

很明显，这些变量名比 count1, abc, commandlinebuffer 和 FILEFLAG 之类的命名要易于理解。由于匈牙利表示法既描述了变量的类型，又描述了变量的作用，所以能帮助程序员及早发现变量的使用错误，如把一个数值当指针来使用引发的内存页错误等。

对于函数名，由于不会返回多种类型的数值，所以命名时一般不再用类型开头，但名称还

是用表示用途的单词组成，每个单词的首字母大写。Windows API 是这种命名方式的绝好例子，当人们看到 ShowWindow, GetWindowText, DeleteFile 和 GetCommandLine 之类的 API 函数名称时，恐怕不用查手册，就能知道它们是做什么用的。比起 int 21h/09h 和 int 13h/02h 之类的中断调用，好处是不必多讲的。

2. 对匈牙利表示法的补充

使用匈牙利表示法已经基本上解决了命名的可读性问题，但相对于其他高级语言，汇编语言有语法上的特殊性，考虑下面这些汇编语言特有的问题：

- 对局部变量的地址引用要用 lea 指令或用 addr 伪操作，全局变量要用 offset；对局部变量的使用要特别注意初始化问题。如何在定义中区分全局变量、局部变量和参数？
- 汇编的源代码占用的行数比较多，代码行数很容易膨胀，程序规模大了如何分清一个函数是系统的 API 还是本程序内部的子程序？

实际上上面的这些问题都可以归纳为区分作用域的问题。为了分清变量的作用域，命名中对全局变量、局部变量和参数应该有所区别，所以我们需要对匈牙利表示法做一些补充，以适应 Win32 汇编的特殊情况，下面的补充方法是笔者提出的，读者可以参考使用：

- 全局变量的定义使用标准的匈牙利表示法，在参数的前面加下划线，在局部变量的前面加@符号，这样引用的时候就能随时注意到变量的作用域。
- 在内部子程序的名称前面加下划线，以便和系统 API 区别。

如下面是一个求复数模的子程序，子程序名前面加下划线表示这是本程序内部模块，两个参数——复数的实部和虚部用 _dwX 和 _dwY 表示，中间用到的局部变量 @dwResult 则用 @号开头：

```

_Calc    proc _dwX, _dwY
          local    @dwResult

          finit
          fild _dwX
          fld     st(0)
          fmul                    ;i * i
          fild _dwY
          fld     st(0)
          fmul                    ;j * j
          fadd                    ;i * i + j * j
          fsqrt                    ;sqrt(i * i + j * j)
          fistp   @dwResult ;put result
          mov     eax, @dwResult
          ret

_Calc    endp

```

本书中所有的示范源代码采用的都是这样的命名约定。

3.6.2 代码的书写格式

1. 排版方式

程序的排版风格应该遵循以下规则。

首先是大小写的问题，汇编程序中对于指令和寄存器的书写是不分大小写的，但小写代码比大写代码便于阅读，所以程序中的指令和寄存器等最好采用小写字母，而用 `equ` 伪操作符定义的常量则使用大写，变量和标号使用匈牙利表示法，大小写混合。

其次是使用 `Tab` 的问题。汇编源程序中 `Tab` 的宽度一般设置为 8 个字符。在语法上，指令和操作数之间至少有一个空格就可以了，但指令的助记符长度是不等长的，用 `Tab` 隔开指令和操作数可以使格式对齐，便于阅读。如：

```
xor  eax, eax
fistp dwNumber
xchg  eax, ebx
```

上述代码的写法就不如下面的写法整齐：

```
xor      eax, eax
fistp    dwNumber
xchg     eax, ebx
```

还有就是缩进格式的问题。程序中的各部分采用不同的缩进，一般变量和标号的定义不缩进，指令用两个 `Tab` 缩进，遇到分支或循环伪指令再缩进一格，如：

```
dwFlag      .data
             dd      ?
start:       .code
             mov     eax, dwFlag
             .if     dwFlag == 1
               call  _Function1
             .else
               call  _Function2
             .endif
             ...
```

合适的缩进格式可以明显地表现出程序的流程结构，也很容易发现嵌套错误，当缩进过多的时候，可以意识到嵌套过深，该改进程序结构了。

2. 注释和空行

没有注释的程序是很难维护的，但注释的方法也很有讲究，写注释要遵循以下的规则：

- 不要写无意义的注释，如“将 1 放到 `eax` 中”，“跳转到 `exit` 标号处”等。
- 修改代码同时修改相应的注释，以保证注释与代码的一致性。
- 注释以描写一组指令实现的功能为主，不要解释单个指令的用法，那是应该由指令手册来完成的，不要假设看程序的人连指令都不熟悉。

- 对于子程序，要在头部加注释说明参数和返回值，子程序可以实现的功能，以及调用时应该注意的事项。

由于汇编语言是以一条指令为一行的，实现一个小功能就需要好几行，没有分段的程序很难看出功能模块来，所以要合理利用空行来隔开不同的功能块，一般以在高级语言中可以用一句语句来完成的一段汇编指令为单位插入一个空行。

3. 避免使用宏

在 MASM 的宏功能中最好只使用条件汇编，用来选择编译不同的代码块来构建不同的版本，其他如宏定义和宏调用只会破坏程序的可读性，能够不用就尽量不用，虽然展开后只有一两句的宏定义不在此列，但既然展开后也只是一两句，那么和直接使用指令也就没有什么区别了。

在汇编中避免使用宏定义的理由是：汇编中随时要用到各个寄存器，宏定义不同于子程序，可以有选择地通过 push/pop 指令保护现场，在使用中很容易忽略里面用了哪个寄存器，从而对程序结构构成威胁。高级语言的宏定义则不会有这个问题。

笔者曾经见到过最极端的使用宏定义的程序是 MicroMedia 的 Director SDK，100 行左右的例子中几乎有 90% 都是宏定义，虽然例子很容易改成其他功能的程序，但要在里面加入新的功能则几乎是不可能的，因为程序中连 C 语言函数开始和结束的花括号都被改成了宏定义，这样一来，如果要真正使用这个开发包，则必须把宏定义“翻译”回原来的样子才能真正理解程序的流程。对于这样的代码，笔者是绝对不敢苟同的。

3.6.3 代码的组织

程序中要注意变量的组织和模块的组织方式。

由于过多的全局变量会影响程序的模块化结构，所以不要设置没必要的全局变量，尽量把变量定义成局部变量。把仅在子程序中使用的变量设置为局部变量可以使子程序更容易封装成一个黑匣子，如果无法把全部变量设置为局部变量，则尽量把这些数据改为参数输入输出，如果无法改为参数，那么意味着这个子程序不能不经修改地直接放到别的程序中使用。

在主程序中使用比较频繁的部分，以及便于封装成黑匣子在别的程序上用的代码，都应该写成子程序，但一个子程序的规模不应该太大，行数尽量限制在几百行之内，功能则限于完成单个功能。对于子程序，定义参数的时候要尽可能精简，对可能引起程序崩溃的参数，如指针等，要进行合法性检测。

子程序中在使用完申请的资源的时候，注意在退出前要释放所用资源，包括申请的内存和其他句柄等，对于打开的文件则要关闭。

对于程序员来说，开发每一个软件都要从头做起是很浪费时间的，一般的做法是从自己以前做过的程序中拷贝相似的代码，但修改还是要花一定的时间，最好的办法就是尽量把子程序做成一个黑匣子，可以不经修改地直接拿过来用，这样，每次编程相当于只是编写新增的部分，随着代码的积累，开发任何程序都将是很快的事情。

第 4 章

第一个窗口程序

4.1 开始了解窗口

4.1.1 窗口是什么

窗口是什么？大家每天在使用 Windows，屏幕上的一个个方块就是一个个窗口！那么，窗口为什么是这个样子呢？窗口就是程序吗？

1. 使用窗口的原因

回想一下 DOS 时代的计算机屏幕，在 1990 年 Windows 3.0 推出之前，计算机的屏幕一直使用文本模式，黑洞洞的底色上漂浮着白色的小字，性能不高的图形模式只用于简单的游戏和一些图形软件。对 DOS 程序来说，屏幕是惟一的，上面有个光标表示输入字符的位置，程序运行后往屏幕输出一些信息，退出时输出的信息就留在了屏幕上，然后是第二个程序重复这个过程，当屏幕被写满的时候，整个屏幕上卷一行，最上面一行被去掉，然后程序在最底下新空出来的一行上继续输出。

对于一个单任务的操作系统来说，这种方式是很合理的，因为平时使用的传真机或打字机就是用上卷的方式来容纳新的内容的。但是如果是多任务呢？两个程序同时往屏幕上输出字符或者两个人同时往打字机上打字，那么谁都看不懂混在一起的是什么。DOS 下的 TSR（内存驻留）程序是多个程序同时使用一个屏幕的例子，但实质上这并不是多任务，而是 TSR 将别的程序暂时挂起，挂起的程序不可能在 TSR 执行期间再向屏幕输出内容，TSR 在输出自己的内容之前必须保存屏幕上显示的内容，并在退出的时候把屏幕恢复原来的样子，否则挂起的程序并不知道屏幕已经被改变，在这个过程中，DOS 不会去干预中间发生的一切。

Windows 是多任务的操作系统，可以同时运行多个程序，同样，各个程序在屏幕上的显示不能互相干扰，而且，多个程序可以看成是“同时”运行的，在后台的程序也可能随时向屏幕输出内容，这中间的调度是由 Windows 完成的。Windows 采用的方法是给程序一块矩形的屏幕空间，这就是窗口。应用程序通过 Windows 向属于自己的窗口显示信息，Windows 判断该窗口是不是被别的窗口挡住，并把没有挡住的部分输出到屏幕上，这样屏

幕上显示的窗口就不会互相覆盖而乱套。对于应用程序来说，它只需认为窗口就是自己拥有的显示空间就可以了。

2. 窗口和程序的关系

既然不同窗口的内容就是不同程序的输出，那么一个窗口就是一个程序吗？反过来，一个程序就是一个窗口吗？

答案是否定的，一个窗口不一定就是一个程序，它可能只是一个程序的一部分。一个程序可以建立多个顶层窗口，如 Windows 的桌面和任务栏都是顶层窗口，但它们都属于“文件管理器”进程，所以并不是一个窗口就是一个程序的代表。Windows 的窗口采用层次结构，一个窗口中可以建立多个子窗口，如窗口中的状态栏、工具栏，对话框中的按钮、文本输入框与复选框等都是子窗口。子窗口中还可以再建立下一级子窗口，如 Word 工具栏上的字体选择框。

反过来，运行的程序并非一定就是窗口，比如悄悄在后台运行的木马程序就不会显示一个窗口向用户报告它在干非法勾当。在 Windows NT 下用“任务管理器”查看，进程的数量比屏幕上的窗口多得多，意味着很多的运行程序并没有显示窗口。如果一个程序不想和用户交互，它可以选择不建立窗口。

所以本章的标题“第一个窗口程序”指学习编写第一个以标准的窗口为界面的程序，而不是泛指 Windows 程序。如果要写的 Win32 程序不是以窗口为界面的（如控制台程序等），就不一定采用本章中提及的以消息驱动的程序结构。

虽然以窗口为界面的程序并不是所有 Windows 程序的必然选择，但绝大部分的应用程序是以这种方式出现的，从操作系统的名称“Windows”就可以看出这一点，了解窗口程序就是相当于在了解 Windows 工作方式的基础。

控制台方式也是 Windows 程序的另一种常用界面，考虑到初学者刚刚接触 Windows 程序的体系架构，将控制台界面编程的内容插在本章或者后续章节中容易引起初学者对窗口程序架构理解上的混淆，所以本书将控制台编程单独放在附录 A 中（以电子版方式放在随书光盘中），有兴趣的读者可以在学完资源、图形编程、界面编程等内容后再单独阅读这个章节。

4.1.2 窗口界面

大部分的窗口看上去都是大同小异的，先来看一个典型的窗口——Windows 附带的写字板，该程序的界面如图 4.1 所示，我们将用它来说明窗口的各个组成部分。

窗口一般由屏幕上的矩形区域组成，不同的窗口可能包括一些相同的组成部分，如标题栏、菜单、工具栏、边框和状态栏等，每个部分都有自己固定的行为模式：

- 窗口边框——窗口的外沿就是窗口边框，用鼠标按住边框并拖动可以调整窗口的大小。
- 标题栏——窗口的最上面是标题栏，用鼠标按住标题栏拖动可移动窗口，双击标题栏则将窗口最大化或从最大化的状态恢复。通过标题栏的颜色可以区分窗口是不是活动窗口，同时标题栏列出了应用程序的名称。



图 4.1 一个典型的窗口

- 菜单——标题栏下面是菜单，单击菜单会弹出各种功能选择。
- 工具栏——菜单的下面是工具栏，工具栏上用图标的方式列出最常用的功能，相当于菜单的快捷方式。
- 图标和“最小化”、“最大化”与“关闭”按钮——图标位于标题栏的左边，三个控制按钮则位于标题栏的右边。单击图标会弹出一个系统菜单，双击图标则相当于按下了“关闭”按钮。“最小化”、“最大化”按钮用来控制窗口的大小。
- 状态栏——状态栏位于窗口的最下面，用来显示一些状态信息。
- 客户区——窗口中间用来工作或输出的区域叫做窗口的客户区，把窗口看做是一张白纸的话，客户区就是白纸中真正用来书写的部分，程序在这里和用户进行交互。
- 滚动条——如果客户区太小不足以显示全部内容，则右边或底部可能还有滚动条，拖动它可以滚动窗口的客户区，以便看到其他的内容。

虽然大部分窗口看上去都差不多，但并不是每个窗口都有这些组成部分，也许有的窗口就没有图标和最小化、最大化框，有的没有工具栏或状态栏，有的没有标题栏，而有的就干脆是个奇怪的形状，如 Office 帮助中的助手，那些小狗小猫都是些不折不扣的窗口，本书第 7 章中的 BmpClock 例子就是类似的不规则窗口的例子，另外，Windows 的桌面和桌面下面的任务栏也都是窗口，就连屏幕保护程序的黑屏幕也是一个大小为整个屏幕、没有标题栏和边框的窗口！

一致的窗口形状和行为模式为 Windows 用户提供了一致的用户界面，几乎所有的窗口程序都在菜单的第一栏设置有关文件的操作和退出功能、最后一栏设置程序的帮助，相同的功能在工具栏上的图标也是大同小异的，用户可以不再像在 DOS 下那样，对不同的程序需要学习不同的界面，用户自从学会使用第一个软件起，就基本学会了所有 Windows 软件的使用模式，而且可以通过相似的菜单、工具栏等来发掘程序的新功能。窗口的菜单和客户区是最个性化的部分，菜单随程序功能的不同而不同，而客户区则是窗口程序的输出区域，不同的程序在客户区内显示了不同的内容。

4.1.3 窗口程序是怎么工作的

1. 窗口程序的运行模式

对程序员来说，要了解的不仅是用户可以看到的部分，还必须了解隐藏在窗口底下的细节，了解用怎样的程序结构来实现窗口的行为模式。

DOS 程序员熟悉的是顺序化的、按过程驱动的程序设计方法，这种程序有明显的开始、明显的过程和明显的结束，由程序运行的阶段来决定用户该做什么。

而窗口程序是事件驱动的，用户可能随时发出各种消息，如操作的过程中觉得窗口不够大了，就马上拖动边框，程序必须马上调整客户区的内容以适应新的窗口大小；用户觉得想先干别的事情，可能会把窗口最小化，关闭按钮也有可能随时被按下，这意味着程序要随时可以处理退出的请求。如果非要规定干活的时候不能移动窗口与调整大小，那么这些窗口就会呆在桌面上一动不动。



再次提醒：这里是“窗口程序”而不是“Windows 程序”，因为和窗口有关的程序才是事件驱动的，其他的 Windows 可能并不这样工作，如控制台程序的结构还是同 DOS 程序一样是顺序化的，但与窗口相关的 Windows 程序占了绝大多数，所以大部分书籍中讲到 Windows 程序就认为是事件驱动的程序。

先通过一个简单的例子来说明两种程序设计方式的不同，以 DOS 下的文件比较命令 `comp` 为例，程序运行时先提示输入第一个文件名，然后是输入第二个文件名，程序比较后退出，同时把结果输出在屏幕上。假如有一个窗口版的 `comp` 程序，那么运行时会在屏幕上出现一个对话框，上面有两个文本框用来输入两个文件名，还会有个“比较”按钮，按下后开始比较文件，用户可以随时按下“关闭”按钮来退出程序。

两种程序的运行会有相当大的不同，如图 4.2 所示，DOS 程序必须按照顺序运行，当运行到输入第二个文件名时，用户不可能回到第一步修改第一个文件名，这时候用户也不能退出（除非用户强制用 `Ctrl+C` 键，但这不是程序的本意）；而在窗口程序中用户可以随意选择先输入哪个文件名，同时也可以对窗口进行各种操作，当用户做任何一个操作的时候，相当于发出了一个消息，这些消息没有任何顺序关系，程序中必须随时准备处理不同的消息。

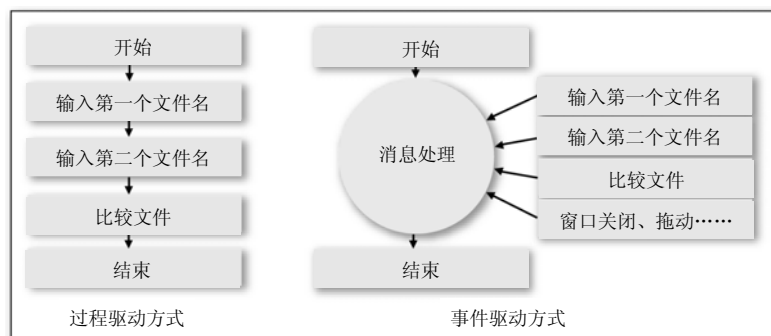


图 4.2 不同的程序结构模式

力
斤

力
斤

力
斤

力
斤

力
斤

94

接下来开始分析源代码,看了这近三页的源代码,第一个感觉是什么?是不是想撤退了?笔者刚开始编 Win32 程序的时候就是这种感觉,可能 90% 的人有同样的感觉,别急,过了这一关,Win32 汇编的入门就成功了一半,所以千万要挺住!有个振奋人心的消息是,这个程序是大部分窗口程序的模板,以后要写一个新的程序,把它拷贝过来再往中间添砖加瓦就是了,工夫一点都



分析一下程序的结构，发现入口是 start，然后执行了一个 WinMain 子程序，完成后就是程序退出的函数 ExitProcess，再看 WinMain 的结构，前面是顺序下来的几个 API：

接下来，就是一个由 3 个 API 组成的循环了：

很明显，这是与消息有关的循环，因为 API 名称中都带有 Message 字样，如果退出这个循环，程序也就结束了，这个循环叫做消息循环。设置_WinMain 子程序并不是必须的，可以把_WinMain 的所有代码放到主程序中，没有任何影响，之所以这样只是为了将这里使用的变量定义成局部变量，这样可以方便移植。

看了程序的流程，似乎没有什么地方涉及窗口的行为，如改变大小和移动位置的处理等。再看源程序，除了_WinMain，还有一个子程序_ProcWinMain，但除了在 WNDCLASSEX 结构的赋值中提到过它，好像就没有什么地方要用到这个子程序，起码在自己编写的源代码中没有任何一个地方调用过它。

再看_ProcWinMain，它是一个分支结构处理的子程序，功能是把参数 uMsg 取出来，根据不同的 uMsg 执行不同的代码，完了以后就退出了，中间也没有任何代码和主程序有关联。

第一个窗口程序就是由这么两个似乎是风马牛不相及的部分组成的，但它确实能工作，对于写惯了 DOS 汇编的程序员来说，这似乎不可理解。下面来看看这么一个陌生而奇怪的程序是如何工作的。

3. 窗口程序的运行过程

在屏幕上显示一个窗口的过程一般有以下步骤，这就是主程序的结构流程：

- (1) 得到应用程序的句柄 (GetModuleHandle)。
- (2) 注册窗口类 (RegisterClassEx)。在注册之前，要先填写 RegisterClassEx 的参数 WNDCLASSEX 结构。
- (3) 建立窗口 (CreateWindowEx)。
- (4) 显示窗口 (ShowWindow)。
- (5) 刷新窗口客户区 (UpdateWindow)。
- (6) 进入无限的消息获取和处理的循环。首先获取消息 (GetMessage)，如果有消息到达，则将消息分派到回调函数处理 (DispatchMessage)，如果消息是 WM_QUIT，则退出循环。

程序的另一半_ProcWinMain 子程序是用来处理消息的，它就是窗口的回调函数 (Callback)，也叫做窗口过程，之所以是回调函数是因为它是由 Windows 而不是我们自己调用的，我们调用 DispatchMessage，而 DispatchMessage 在自己的内部反过来调用窗口过程。

所有的用户操作都是通过消息来传给应用程序的，如用户按键、鼠标移动、选择了菜单和拖动了窗口等，应用程序中由窗口过程接收消息并处理，在例子程序中就是_ProcWinMain。由于窗口过程构造了一个分支结构，对应不同的消息执行不同的代码，所以一个应用程序中几乎所有的功能代码都集中在窗口过程里。

窗口程序运行中消息传输的流程可以由图 4.4 来表示。

先来看看 Windows 对消息的处理。Windows 在系统内部有一个系统消息队列，当输入设备有所动作的时候，如用户按动了键盘、移动了鼠标、按下或放开了鼠标等，Windows 都会产生相应的记录放在系统消息队列里，如图 4.4 中的箭头 a 和 b 所示，每个记录中包含消息的类型、发生的位置（如鼠标在什么坐标移动）和发生的时间等信息。

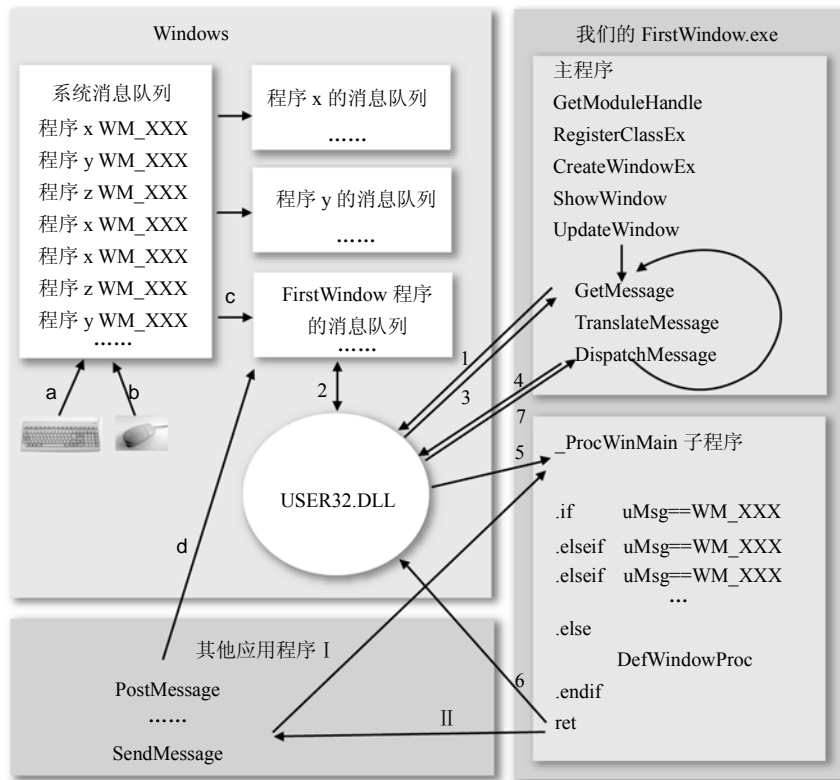


图 4.4 窗口程序的运行过程

同时，Windows 为每个程序（严格地说是每个线程）维护一个消息队列，Windows 检查系统消息队列里消息的发生位置，当位置位于某个应用程序的窗口范围内的时候，就把这个消息派送到应用程序的消息队列里，如图 4.4 中的箭头 c 所示。

当应用程序还没有来取消息的时候，消息就暂时保留在消息队列里，当程序中的消息循环执行到 `GetMessage` 的时候，控制权转移到 `GetMessage` 所在的 `USER32.DLL` 中（箭头 1），`USER32.DLL` 从程序消息队列中取出一条消息（箭头 2），然后把这条消息返回应用程序（箭头 3）。

应用程序可以对这条消息进行预处理，如可以用 `TranslateMessage` 把基于键盘扫描码的按键消息转换成基于 ASCII 码的键盘消息，以后也会用到 `TranslateAccelerator` 把键盘快捷键转换成命令消息，但这个步骤不是必需的。

然后应用程序将处理这条消息，但方法不是自己直接调用窗口过程来完成，而是通过 `DispatchMessage` 间接调用窗口过程，`Dispatch` 的英文含义是“分派”，之所以是“分派”，

是因为一个程序可能建有不只一个窗口，不同的窗口消息必须分派给相应的窗口过程。当控制权转移到 USER32.DLL 中的 `DispatchMessage` 时，`DispatchMessage` 找出消息对应窗口的窗口过程，然后把消息的具体信息当做参数来调用它（箭头 5），窗口过程根据消息找到对应的分支去处理，然后返回（箭头 6），这时控制权回到 `DispatchMessage`，最后 `DispatchMessage` 函数返回应用程序（箭头 7）。这样，一个循环就结束了，程序又开始新一轮的 `GetMessage`。



有个很常见的问题：为什么要由 Windows 来调用窗口过程，程序取了消息以后自己处理不是更简便吗？事实上并非如此，如果程序自己处理消息的“分派”，就必须自己维护本程序所属窗口的列表，当程序建立的窗口不止一个的时候，这个工作就变得复杂起来；另一个原因是：别的程序也可能用 `SendMessage` 通过 Windows 直接调用你的窗口过程；第三个原因：Windows 并不是把所有的消息都放进消息队列，有的消息是直接调用窗口过程处理的，如 `WM_SETCURSOR` 等实时性很强的消息，所以窗口过程必须开放给 Windows。

应用程序之间也可以互发消息，`PostMessage` 是把一个消息放到其他程序的消息队列中，如图 4.4 中箭头 d 所示，目标程序收到了这条消息就把它放入该程序的消息队列去处理；而 `SendMessage` 则越过消息队列直接调用目标程序的窗口过程（如图 4.4 中箭头 I 所示），窗口过程返回以后才从 `SendMessage` 返回（如图 4.4 中箭头 II 所示）。

窗口过程是由 Windows 回调的，Windows 又是怎么知道往哪里回调呢？答案是我们在调用 `RegisterClassEx` 函数的时候已经把窗口过程的地址告诉了 Windows。

4.2 分析窗口程序

了解了消息驱动体系的工作流程以后，让我们来分析如何用 Win32 汇编实现这一切，本节和下一节将详细分析 `FirstWindow` 源程序。

4.2.1 模块和句柄

1. 模块的概念

一个模块代表的是一个运行中的 `exe` 文件或 `dll` 文件，用来代表这个文件中所有的代码和资源，磁盘上的文件不是模块，装入内存后运行时就叫做模块。一个应用程序调用其他 DLL 中的 API 时，这些 DLL 文件被装入内存，就产生了不同的模块，为了区分地址空间中的不同模块，每个模块都有一个惟一的模块句柄来标识。

由于很多 API 函数中都要用到程序的模块句柄，以便利用程序中的各种资源，所以在程序的一开始就先取得模块句柄并存放到一个全局变量中可以省去很多的麻烦，在 Win32 中，模块句柄在数值上等于程序在内存中装入的起始地址。

取模块句柄使用的 API 函数是 `GetModuleHandle`，它的使用方法是：

```
invoke    GetModuleHandle, lpModuleName
```

`lpModuleName` 参数是一个指向含有模块名称字符串的指针，可以用这个函数取得程序地址空间中各个模块的句柄，例如，如果想得到 `User32.dll` 的句柄以便使用其中包含的图标资源，那么可以如下使用：

```
szUserDll db      'User32.dll', 0
...
        invoke    GetModuleHandle, addr szUserDll
        .if      eax
            mov     hUserDllHandle, eax
        .endif
...
```

如果使用参数 `NULL` 调用 `GetModuleHandle`，那么得到的是调用者本模块的句柄，我们的源程序中就是这样使用的：

```
invoke    GetModuleHandle, NULL
mov       hInstance, eax
```

可以注意到，把返回的句柄放到了取名为 `hInstance` 的变量里而并不是放在 `hModule` 中，为什么是 `hInstance` 呢？`Instance` 是“实例”，它的概念来自于 Win16，Win16 中不同运行程序的地址空间并非是完全隔离的，一个可执行文件运行后形成“模块”，多次加载同一个可执行文件时，这个“模块”是公用的，为了区分多次加载的“拷贝”，就把每个“拷贝”叫做实例，每个实例均用不同的“实例句柄”（`hInstance`）值来标识它们。

但在 Win32 中，程序运行时是隔离的，每个实例都使用自己私有的 4 GB 空间，都认为自己是惟一的，不存在一个模块的多个实例的问题，实际上在 Win32 中，实例句柄就是模块句柄，但很多 API 原型中用到模块句柄的时候使用的名称还是沿用 `hInstance`，所以我们还是把变量名称取为 `hInstance`。



在 C 语言的编程中，`hInstance` 通过 `WinMain` 由系统传入，`WinMain` 的原型是：
`WinMain (hInstance, hPrevInstance, lpzCmdParam, nCmdShow)`，程序不用自己去获得 `hInstance`，这个过程由 C 的初始化代码代劳了，但在 Win32 汇编中 `hInstance` 必须自己获取，如果不了解 `hModule` 就是 `hInstance` 的话，就无法得知如何得到 `hInstance`，因为并没有一个类似于 `GetInstanceHandle` 之类的 API 函数。

2. 句柄是什么

随着分析的深入，句柄（`handle`）一词也出现得频繁起来，“句柄”是什么呢？句柄只是一个数值而已，它的值对程序来说是没有意义的，它只是 Windows 用来表示各种资源的编号而已，可见只有 Windows 才知道怎么使用它来引用各种资源。

下面举例说明。屏幕上已经有 10 个窗口，Windows 把它们从 1 到 10 编号，应用程序

又建立了一个窗口，现在 Windows 把它编号为 11，然后把 11 当做窗口句柄返回给应用程序，应用程序并不知道 11 代表的是什么，但在操作窗口的时候，把 11 当做句柄传给 Windows，Windows 自然可以根据这个数值查出是哪个窗口。当该窗口关闭的时候，11 这个编号作废。第二次运行的时候，如果屏幕上现有 5 个窗口，那么现在句柄可能就是 6 了，所以，应用程序并不关心句柄的具体数值是多少。打个比方，可以把句柄当做是商场中寄放书包时营业员给的纸条，纸条上的标记用户并不知道是什么意思，但把它交还给营业员的时候，她自然会找到正确的书包。

Windows 中几乎所有的东西都是用句柄来标识的，文件句柄、窗口句柄、线程句柄和模块句柄等，同样道理，不必关心它们的值究竟是多少，拿来用就是了！

4.2.2 创建窗口

在创建窗口之前，先要谈到“类”。“类”的概念读者都不陌生，主要是为了把一组物体的相同属性归纳整理起来封装在一起，以便重复使用，在“类”已定义的属性基础上加上其他个性化的属性，就形成了各式各样的个体。

Windows 中创建窗口同样使用这样的层次结构。首先定义一个窗口类，然后在窗口类的基础上添加其他的属性建立窗口。不用一步到位的办法是因为很多窗口的基本属性和行为都是一样的，如按钮、文本输入框和选择框等，对这些特殊的窗口 Windows 都预定义了对应的类，使用时直接使用对应的类名建立窗口就可以了。只有用户自定义的窗口才需要先定义自己的类，再建立窗口。这样可以节省资源。

1. 注册窗口类

建立窗口类的方法是在系统中注册，注册窗口类的 API 函数是 RegisterClassEx，最后的“Ex”是扩展的意思，因为它是 Win16 中 RegisterClass 的扩展。一个窗口类定义了窗口的一些主要属性，如：图标、光标、背景色、菜单和负责处理该窗口所属消息的函数。这些属性并不是分成多个参数传递过去的，而是定义在一个 WNDCLASSEX 结构中，再把结构的地址当参数一次性传递给 RegisterClassEx，WNDCLASSEX 是 WNDCLASS 结构的扩展。

WNDCLASSEX 的结构定义为：

WNDCLASSEX	STRUCT		
cbSize	DWORD	?	;结构的字节数
style	DWORD	?	;类风格
lpfnWndProc	DWORD	?	;窗口过程的地址
cbClsExtra	DWORD	?	
cbWndExtra	DWORD	?	
hInstance	DWORD	?	;所属的实例句柄
hIcon	DWORD	?	;窗口图标
hCursor	DWORD	?	;窗口光标
hbrBackground	DWORD	?	;背景色
lpszMenuName	DWORD	?	;窗口菜单
lpszClassName	DWORD	?	;类名字符串的地址
hIconSm	DWORD	?	;小图标
WNDCLASSEX	ENDS		

在 FirstWindow 程序中，注册窗口类的代码是：

```

local    @stWndClass:WNDCLASSEX ; 定义一个 WNDCLASSEX 结构
...

invoke   RtlZeroMemory, addr @stWndClass, sizeof @stWndClass
invoke   LoadCursor, 0, IDC_ARROW
mov      @stWndClass.hCursor, eax
push     hInstance
pop      @stWndClass.hInstance
mov      @stWndClass.cbSize, sizeof WNDCLASSEX
mov      @stWndClass.style, CS_HREDRAW or CS_VREDRAW
mov      @stWndClass.lpfWndProc, offset _ProcWinMain
mov      @stWndClass.hbrBackground, COLOR_WINDOW + 1
mov      @stWndClass.lpszClassName, offset szClassName
invoke   RegisterClassEx, addr @stWndClass

```

程序定义了一个 WNDCLASSEX 结构的变量 @stWndClass，用 RtlZeroMemory 将它填为零（局部变量初始化的重要性在第 3 章中已经强调过），再填写结构的各个字段，这样，没有赋值的部分就保持为 0，结构各字段的含义如下：

- hIcon——图标句柄，指定显示在窗口标题栏左上角的图标。Windows 已经预定义了一些图标，同样，程序也可以使用在资源文件中定义的图标，这些图标的句柄可以用 LoadIcon 函数获得。因为例子程序没有用到图标，所以 Windows 给窗口显示了一个默认的图标。
- hCursor——光标句柄，指定了鼠标在窗口中的光标形状。同样，Windows 也预定义了一些光标，可以用 LoadCursor 获取它们的句柄，IDC_ARROW 是 Windows 预定义的箭头光标，如果想使用自定义的光标，也可以自己在资源文件中定义。
- lpszMenuName——指定窗口上显示的默认菜单，它指向一个字符串，描述资源文件中菜单的名称，如果资源文件中菜单是用数值定义的，那么这里使用菜单资源的数值。窗口中的菜单也可以在建立窗口函数 CreateWindowEx 的参数中指定。如果在两个地方都没有指定，那么建立的窗口上就没有菜单。
- hInstance——指定要注册的窗口类属于哪个模块，模块句柄在程序开始的地方已经用 GetModuleHandle 函数获得。
- cbSize——指定 WNDCLASSEX 结构的长度，用 sizeof 伪操作来获取。很多 Win32 API 参数中的结构都有 cbSize 字段，它主要是用来区分结构的版本，当以后新增了一个字段时，cbSize 就相应增大，如果调用的时候 cbSize 还是旧的长度，表示运行的是基于旧结构的程序，这样可以防止使用无效的字段。
- style——窗口风格。CS_HREDRAW 和 CS_VREDRAW 表示窗口的宽度或高度改变时是否重画窗口。比较重要的是 CS_DBLCLKS 风格，指定了它，Windows 才会把在窗口中快速两次单击鼠标的行为翻译成双击消息 WM_LBUTTONDOWNBLCLK 发给窗口过程。笔者就曾经忘了指定它，结果怎么也搞不出双击消息来。

- hbrBackground——窗口客户区的背景色。前面的 hbr 表示它是一个刷子（Brush）的句柄，“刷子”一词形象地表示了填充一个区域的着色模式。Windows 预定义了一些刷子，如 BLACK_BRUSH 和 WHITE_BRUSH 等，可以用下列语句来得到它们的句柄：

```
invoke    GetStockObject, WHITE_BRUSH
```

但在这里也可以使用颜色值，Windows 已经预定义了一些颜色值，分别对应窗口各部分的颜色，如 COLOR_BACKGROUND, COLOR_HIGHLIGHT, COLOR_MENU 和 COLOR_WINDOW 等，使用颜色值的时候，Windows 规定必须在颜色值上加 1，所以程序中的指令是：

```
mov       @stWndClass.hbrBackground, COLOR_WINDOW + 1
```

- lpszClassName——指定程序员要建立的类命名，以便以后用这个名称来引用它。这个字段是一个字符串指针，在程序里，它指向“MyClass”字符串。
- cbWndExtra 和 cbClsExtra——分别是在 Windows 内部保存的窗口结构和类结构中给程序员预留的空间大小，用来存放自定义数据，它们的单位是字节。不使用自定义数据的话，这两个字段就是 0。
- lpfnWndProc——最重要的参数，它指定了基于这个类建立的窗口的窗口过程地址。通过这个参数，Windows 就知道了在 DispatchMessage 函数中把窗口消息发到哪里去，一个窗口过程可以为多个窗口服务，只要这些窗口是基于同一个窗口类建立的。Windows 中不同应用程序中的按钮和文本框的行为都是一样的，就是因为它们是基于相同的 Windows 预定义类建立的，它们背后的窗口过程其实是同一段代码。

结构中的 style 表示窗口的风格，Windows 已经有一些预定义的值，它们是以 CS(Class Style 的缩写) 开始的标识符，如表 4.1 所示。

表 4.1 一些窗口类的 style 预定义值

预定义值	十六进制值	对应二进制位
CS_VREDRAW	00000001h	0
CS_HREDRAW	00000002h	1
CS_KEYCVTWINDOW	00000004h	2
CS_DBLCLKS	00000008h	3
...

可以看到，这些预定义值实际上是在使用不重复的数据位，所以可以组合起来使用，同时使用不同的预定义值并不会引起混淆。



对于不同二进制位组合的计算，“加”和“或”的结果是一样的，在 FirstWindow 程序中用 CS_HREDRAW or CS_VREDRAW 来代表两个组合，若用 CS_HREDRAW + CS_VREDRAW 也并没有什么不同，但强烈建议使用 or，因为如果不小心指定了两个同样的风格时：CS_HREDRAW or CS_VREDRAW or CS_VREDRAW 和原来的数值是一样的，而 CS_HREDRAW + CS_VREDRAW + CS_VREDRAW 就不对了，因为 1 or 1 = 1，而 1 + 1 就等于 2 了。

2. 建立窗口

接下来的步骤是在已经注册的窗口类的基础上建立窗口，使用“类”的原因是定义窗口的“共性”，建立窗口时肯定还要指定窗口的很多“个性化”的参数——如 WNDCLASSEX 结构中没有定义的外观、标题、位置、大小和边框类型等属性，这些属性是在建立窗口时才指定的。

与注册窗口类时用一个结构传递所有参数不同，建立窗口时所有的属性都是用单个参数的方式传递的，建立窗口的函数是 CreateWindowEx（注意不要写成 CreateWindowsEx），同样，它是 Win16 中 CreateWindow 函数的扩展，主要表现在多了一个 dwExStyle（扩展风格）参数，原因是 Win32 比 Win16 中多了很多种窗口风格，原来的一个风格参数已经不够用了。CreateWindowEx 函数的使用方法是：

```
invoke   CreateWindowEx, dwExStyle, lpClassName, lpWindowName, dwStyle, \
        x, y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam
```

虽然这个函数的参数多达 12 个，但它们很好理解：

- lpClassName——建立窗口使用的类名字符串指针，在 FirstWindow 程序中指向“MyClass”字符串，表示使用“MyClass”类建立窗口，这正是我们自己注册的类，这样一来，这个窗口就有“MyClass”的所有属性，并且消息将被发到“MyClass”中指定的窗口过程中去，当然，这里也可以是 Windows 预定义类名，如编辑框就是“EDIT”。
- lpWindowName——指向表示窗口名称的字符串，该名称会显示在标题栏上。如果该参数空白，则标题栏上什么都没有。
- hMenu——窗口上要出现的菜单的句柄。在注册窗口类的时候也定义了一个菜单，那是窗口的默认菜单，意思是如果这里没有定义菜单（用参数 NULL）而注册窗口类时定义了菜单，则使用窗口类中定义的菜单；如果这里指定了菜单句柄，则不管窗口类中有没有定义都将使用这里定义的菜单；两个地方都没有定义菜单句柄，则窗口上没有菜单。另外，当建立的窗口是子窗口时（dwStyle 中指定了 WS_CHILD），这个参数是另一个含义，这时 hMenu 参数指定的是子窗口的 ID 号（这样可以节省一个参数的位置，因为子窗口不会有菜单）。
- lpParam——这是一个指针，指向一个欲传给窗口的参数，这个参数在 WM_CREATE 消息中可以被获取，一般情况下用不到这个字段。
- hInstance——模块句柄，和注册窗口类时一样，指定了窗口所属的程序模块。
- hWndParent——窗口所属的父窗口，对于普通窗口（相对于子窗口），这里的“父子”关系只是从属关系，主要用来在父窗口销毁时一同将其“子”窗口销毁，并不会把窗口位置限制在父窗口的客户区范围内，但如果要建立的是真正的子窗口（dwStyle 中指定了 WS_CHILD 的时候），这时窗口位置会被限制在父窗口的客户区范围内，同时窗口的坐标（x, y）也是以父窗口的左上角为基准的。

- *x*, *y*——指定窗口左上角位置，单位是像素。默认时可指定为 `CW_USEDEFAULT`，这样 Windows 会自动为窗口指定最合适的位置，当建立子窗口时，位置是以父窗口的左上角为基准的，否则，以屏幕左上角为基准。
- *nWidth*, *nHeight*——窗口的宽度和高度，也就是窗口的大小，同样是以像素为单位的。默认时可指定为 `CW_USEDEFAULT`，这样 Windows 会自动为窗口指定最合适的大小。

窗口的两个参数 `dwStyle` 和 `dwExStyle` 决定了窗口的外形和行为，`dwStyle` 是从 Win16 开始就有的属性，表 4.2 列出了一些常见的 `dwStyle` 定义，它们是一些以 `WS`(Windows Style 的缩写) 为开头的预定义值。

表 4.2 窗口风格的预定义值

预定义值	十六进制值	含 义
<code>WS_OVERLAPPED</code>	00000000h	普通的重叠式窗口
<code>WS_POPUP</code>	80000000h	弹出式窗口（没有标题栏）
<code>WS_CHILD</code>	40000000h	子窗口
<code>WS_MINIMIZE</code>	20000000h	初始状态是最小化的
<code>WS_VISIBLE</code>	10000000h	初始状态是可见的
<code>WS_DISABLED</code>	08000000h	初始状态是被禁止的
<code>WS_MAXIMIZE</code>	01000000h	初始状态是最大化的
<code>WS_BORDER</code>	00800000h	单线条边框
<code>WS_DLGFRAME</code>	00400000h	对话框类型的边框
<code>WS_VSCROLL</code>	00200000h	带垂直滚动条
<code>WS_HSCROLL</code>	00100000h	带水平滚动条
<code>WS_SYSMENU</code>	00080000h	带系统菜单（即带标题栏左上角的图标）
<code>WS_THICKFRAME</code>	00040000h	可以拖动调整大小的边框
<code>WS_MINIMIZEBOX</code>	00020000h	有最小化按钮
<code>WS_MAXIMIZEBOX</code>	00010000h	有最大化按钮

为了容易理解，Windows 也为一些定义取了一些别名，同时，由于窗口的风格往往是几种风格的组合，所以 Windows 也预定义了一些组合值，如表 4.3 所示。

表 4.3 等效的窗口风格预定义值

预定义值	等 效 值
<code>WS_CHILDWINDOW</code>	<code>WS_CHILD</code>
<code>WS_TILED</code>	<code>WS_OVERLAPPED</code>
<code>WS_ICONIC</code>	<code>WS_MINIMIZE</code>
<code>WS_SIZEBOX</code>	<code>WS_THICKFRAME</code>
<code>WS_OVERLAPPEDWINDOW</code>	<code>WS_OVERLAPPED</code> or <code>WS_CAPTION</code> or <code>WS_SYSMENU</code> or <code>WS_THICKFRAME</code> or <code>WS_MINIMIZEBOX</code> or <code>WS_MAXIMIZEBOX</code>
<code>WS_TILEDWINDOW</code>	<code>WS_OVERLAPPEDWINDOW</code>
<code>WS_POPUPWINDOW</code>	<code>WS_POPUP</code> or <code>WS_BORDER</code> or <code>WS_SYSMENU</code>

dwExStyle 是 Win32 中扩展的，它们是一些以 WS_EX_ 开头的预定义值，主要定义了一些特殊的风格，表 4.4 给出了一些最常用的特殊风格。

表 4.4 窗口扩展风格的预定义值

预定义值	十六进制值	含 义
WS_EX_TOPMOST	00000008h	总在顶层的窗口
WS_EX_ACCEPTFILES	00000010h	允许窗口进行鼠标拖放操作
WS_EX_TOOLWINDOW	00000080h	工具窗口（很窄的标题栏）
WS_EX_WINDOWEDGE	00000100h	立体感的边框
WS_EX_CLIENTEDGE	00000200h	客户区立体边框
WS_EX_OVERLAPPEDWINDOW		WS_EX_WINDOWEDGE or WS_EX_CLIENTEDGE
WS_EX_PALETTEWINDOW		WS_EX_WINDOWEDGE or WS_EX_TOOLWINDOW or WS_EX_TOPMOST

用预定义的组合值 WS_EX_PALETTEWINDOW 可以很方便地构成浮在其他窗口前面的工具栏。

下面看几种不同的窗口外形，如图 4.5 所示，窗口 1 是 WS_OVERLAPPED 类型的窗口，只有一个边框，没有控制按钮和图标；窗口 2 同时指定 WS_MAXIMIZEBOX, WS_MINIMIZEBOX 和 WS_SYSMENU，在窗口 1 的基础上多了控制按钮和图标；窗口 3 是 WS_POPUPWINDOW 风格的，这是一个没有标题和控制按钮的弹出式窗口，常见的软件装入时的版权窗口就是这种风格；前面 3 个窗口都不能通过拖动边框改变大小，而窗口 4 指定了 WS_THICKFRAME 风格，可以改变大小，它的边框显得厚了一点；窗口 5 的风格是 WS_OVERLAPPEDWINDOW，是最常见的属性组合；窗口 6 在窗口 5 的基础上指定了 WS_EX_CLIENTEDGE，它的客户区显得有立体感；窗口 7 是个工具栏，指定的是 WS_EX_TOOLWINDOW 风格，可以看到它的标题栏要小得多；窗口 8 指定了 WS_HSCROLL 和 WS_VSCROLL 风格，窗口中多了垂直和水平滚动条。

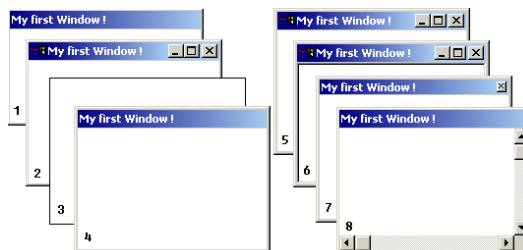


图 4.5 不同风格的窗口

FirstWindow 程序中建立窗口的相关代码是这样的：

```

invoke  CreateWindowEx, WS_EX_CLIENTEDGE, \
        offset szClassName, offset szCaptionMain, \
        WS_OVERLAPPEDWINDOW, \
        100, 100, 600, 400, \
        NULL, NULL, hInstance, NULL
mov     hWinMain, eax
invoke  ShowWindow, hWinMain, SW_SHOWNORMAL
invoke  UpdateWindow, hWinMain
  
```

...

建立窗口以后，eax 中传回来的是窗口句柄，要把它保存起来以备后用，这时候，窗口虽已建立，但还没有在屏幕上显示出来，要用 ShowWindow 把它显示出来，ShowWindow 也可以用在别的地方，主要用来控制窗口的显示状态（显示或隐藏），大小控制（最大化、最小化或原始大小）和是否激活（当前窗口还是背后的窗口），它用窗口句柄做第一个参数，第二个参数则是显示的方式。表 4.5 给出了显示方式预定义值。

表 4.5 ShowWindow 函数显示方式的定义

预定义值	等 效 值
SW_HIDE	隐藏窗口，大小不变，激活状态不变
SW_MAXIMIZE	最大化窗口，显示状态不变，激活状态不变
SW_MINIMIZE	最小化窗口，显示状态不变，激活状态不变
SW_RESTORE	从最大化或最小化恢复正常大小，显示状态不变，激活状态不变
SW_SHOW	显示并激活窗口，大小状态不变
SW_SHOWMAXIMIZED	显示并激活窗口，以最大化显示
SW_SHOWMINIMIZED	显示并激活窗口，以最小化显示
SW_SHOWMINNOACTIVE	显示窗口并最小化，激活状态不变
SW_SHOWNA	显示窗口，大小状态不变，激活状态不变
SW_SHOWNOACTIVATE	显示并从最大化或最小化恢复正常大小，激活状态不变
SW_SHOWNORMAL	显示并激活窗口，恢复正常大小（初始化时用这个参数）

窗口显示以后，用 UpdateWindow 绘制客户区，它实际上就是向窗口发送了一条 WM_PAINT 消息。到此为止，一个顶层窗口就正常建立并显示了。

CreateWindowEx 也可以用来建立子窗口，Windows 中有很多预定义的子窗口类，如按钮和文本框的类名分别是“Button”和“Edit”。要建立一个按钮，只要把 lpClassName 指向“Button”字符串就可以了。下面举例说明建立一个按钮的方法，代码如下：

	.data	
szButton	db	'button',0
szButtonText	db	'&OK',0
	...	
	invoke	CreateWindowEx,NULL,\
		offset szButton,offset szButtonText,\
		WS_CHILD or WS_VISIBLE,\
		10,10,65,22,\
		hWnd,1,hInstance,NULL

在 FirstWindow 的源程序中加入按钮类的定义字符串 szButton 和按钮文字字符串 szButtonText，然后在窗口过程的 WM_CREATE 消息中加入建立按钮的代码，执行一下，窗口中就出现了一个按钮，如图 4.6 所示。建立按钮的时候，lpWindowName 参数就是按钮上的文字，风格则一定要指定 WS_CHILD，建立的按钮才会在我们的主窗口上，WS_VISIBLE 也要同时指定，否则按钮不会显示出来，hMenu 参数在这里用做表示子窗口 ID，将它设置为 1，在建立多个子窗口的时候，ID 应该有所区别。这个例子的源程序可以在所附带光盘

的 Chapter04\FirstWindow-1 目录中找到。



图 4.6 用 CreateWindowEx 建立的按钮

4.2.3 消息循环

1. 消息循环的一般形式

程序中的以下代码就是通常的消息循环：

```
.while    TRUE
        invoke    GetMessage, addr @stMsg, NULL, 0, 0
        .break    .if eax == 0
        invoke    TranslateMessage, addr @stMsg
        invoke    DispatchMessage, addr @stMsg
.endw
```

消息循环中的几个函数要用到一个 MSG 结构，用来做消息传递：

```
MSG STRUCT
    hwnd      DWORD      ?
    message   DWORD      ?
    wParam    DWORD      ?
    lParam    DWORD      ?
    time      DWORD      ?
    pt        POINT      <>
MSG ENDS
```

它的各个字段的含义是：

- hwnd——消息要发向的窗口句柄。
- message——消息标识符，在头文件中以 WM_开头的预定义值（意思为 Windows Message）。
- wParam——消息的参数之一。
- lParam——消息的参数之二。
- time——消息放入消息队列的时间。
- pt——这是一个 POINT 数据结构，表示消息放入消息队列时的鼠标坐标。

这个结构定义了消息的所有属性，GetMessage 函数就是从消息队列中取出这样一条消息的：

```
invoke    GetMessage, lpMsg, hWnd, wParamFilterMin, wParamFilterMax
```

函数的 lpMsg 指向一个 MSG 结构，函数会在这里返回取到的消息，hWnd 参数指定要获取哪个窗口的消息，例子中指定为 NULL，表示获取的是所有本程序所属窗口的消息，

wMsgFilterMin 和 wMsgFilterMax 为 0 表示获取所有编号的消息。

GetMessage 函数从消息队列里取得消息，填写好 MSG 结构并返回，如果获取的消息是 WM_QUIT 消息，那么 eax 中的返回值是 0，否则 eax 返回非零值，所以用 .break .if eax == 0 来检查返回值，如果消息队列中有 WM_QUIT 则退出消息循环。

TranslateMessage 将 MSG 结构传给 Windows 进行一些键盘消息的转换，当有键盘按下和放开时，Windows 产生 WM_KEYDOWN 和 WM_KEYUP 或 WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息，但这些消息的参数中包含的是按键的扫描码，转换成常用的 ASCII 码要经过查表，很不方便，TranslateMessage 遇到键盘消息则将扫描码转换成 ASCII 码并在消息队列中插入 WM_CHAR 或 WM_SYSCHAR 消息，参数就是转换好的 ASCII 码，如此一来，要处理键盘消息的话只要处理 WM_CHAR 消息就好了。遇到非键盘消息则 TranslateMessage 不做处理。

最后，由 DispatchMessage 将消息发送到窗口对应的窗口过程去处理。窗口过程返回后 DispatchMessage 函数才返回，然后开始新一轮消息循环。

2. 其他形式的消息循环

GetMessage 函数是程序空闲的时候主动将控制权交还给 Windows 的一种方式，Windows 是一个抢占式的多任务系统，任务之间每 20 ms 切换一次，试想一下，如果窗口程序在主窗口中采用死循环等待，消息由 Windows 直接发送到窗口过程，那么程序会是下列这种样子：

```

invoke  CreateWindow, ...
invoke  ShowWindow, ...
invoke  UpdateWindow, ...
.while  dwQuitFlag == 0      ;要退出时在窗口过程中设置 dwQuitFlag
.endw
invoke  ExitProcess, ...

```

但这样一来，即使程序在空闲状态，轮到自己的 20 ms 时间片的时候，CPU 时间就会全部消耗在 .while 循环中，使用 GetMessage 的时候，轮到应用程序时间片的时候，如果消息队列里还没有消息，那么程序还是停留在 GetMessage 内部，这时就可以由 Windows 当家做主没收这 20 ms 的时间片，这样保证了 CPU 资源的合理应用。

如果应用程序想把所有时间充分利用回来，消息队列里没有消息的时候不让 GetMessage 在 Windows 内部等待，拱手交出属于自己的 CPU 时间，那么消息循环可以是下列这种样子：

```

.while  TRUE
  invoke  PeekMessage, addr @stMsg, NULL, 0, 0, PM_REMOVE
  .if     eax
    .break  .if @stMsg.message == WM_QUIT
    invoke  TranslateMessage, addr @stMsg
    invoke  DispatchMessage, addr @stMsg
  .else
    <做其他工作>
  .endif
.endw

```

PeekMessage 是一个类似于 GetMessage 的函数,区别在于当消息队列里有消息的时候, PeekMessage 取回消息,并在 eax 中返回非零值,没有消息的时候它会直接返回,并在 eax 中返回零。所以在返回非零值的时候,程序检查消息是否是 WM_QUIT,是则结束消息循环,不是则用标准流程处理消息;返回零的时候,表示是空闲时间,程序就可以做其他工作了,但插入做其他工作的代码执行时间不能过长,以不超过 10 ms 为好,否则会影响正常的消息处理,使窗口的反应看起来很迟钝。如果必须处理很长时间的的工作,那么应该将它分成很多小部分处理,以便有足够的频率用 PeekMessage 来检查消息。

PeekMessage 的前面 4 个参数和 GetMessage 是相同的,增加的最后一个参数表示在取回消息以后,对消息队列中的消息是否保留。当这个参数是 PM_REMOVE 时,消息被取回的同时也被从消息队列里删除,而用 PM_NOREMOVE 的时候,被取回的消息不会从消息队列中删除,函数相当于“偷看”了这条消息。例子程序中用了 PM_REMOVE,否则每次看到的都是队列中的第一条消息。

4.2.4 窗口过程

窗口过程是给 Windows 回调用的,它必须遵循规定的格式。对窗口过程的子程序名并没有规定,对 Windows 来说,窗口过程的地址才是惟一需要的,例子程序中的子程序名是 _ProcWinMain,读者可以改用任何名称。窗口过程子程序的参数格式为:

WindowProc	proc hwnd, uMsg, wParam, lParam
------------	---------------------------------

第一个参数是窗口句柄,由于一个窗口过程可能为多个基于同一个窗口类的窗口服务,所以 Windows 回调的时候必须指出要操作的窗口,否则窗口过程不知道要去处理哪个窗口,FirstWindow 程序只建立了一个窗口,所以每次传递过来的 hwnd 和用 CreateWindowEx 函数返回的窗口句柄是一样的;第二个参数是消息标识,后面两个参数是消息的两个参数。这 4 个参数和消息循环中 MSG 结构中的前 4 个字段是一样的。

1. 窗口过程的结构

窗口过程一般有如下的结构:

WindowProc	proc uses ebx edi esi hWnd, uMsg, wParam, lParam
	<pre> mov eax, uMsg .if eax == WM_XXX <处理 WM_XXX 消息> .elseif eax == WM_YYY <处理 WM_YYY 消息> .elseif eax == WM_CLOSE invoke DestroyWindow, hWinMain invoke PostQuitMessage, NULL .else invoke DefWindowProc, hWnd, uMsg, wParam, lParam ret .endif xor eax, eax ret </pre>

```
WindowProc     endp
```

该过程主要是对 `uMsg` 参数中的消息编号构成一个分支结构，对于需要处理的消息分别处理。不感兴趣的消息则交给 `DefWindowProc` 来处理。

要注意的是，窗口过程中要注意保存 `ebx`, `edi`, `esi` 和 `ebp` 寄存器，高级程序中不用自己操心这一点，汇编中就要注意了，Windows 内部将这 4 个寄存器当指针使用，如果返回时改变了它们的值，程序会马上崩溃。`proc` 后面的 `uses` 伪操作在子程序进入和退出时自动安插上 `push` 和 `pop` 寄存器指令，来保护这些寄存器的值。其实不仅是在窗口过程中是这样，所有由应用程序提供给 Windows 的回调函数都必须遵循这个规定，如定时器回调函数等，所有 Win32 API 也遵循这个规定，所以调用 API 后，`ebx`, `edi`, `esi` 和 `ebp` 寄存器的值总是不会被改变的，但 `ecx` 和 `edx` 的值就不一定了。

`uMsg` 参数指定的消息有一定的范围，Windows 标准窗口中已经预定义的值在 `0~03ffh` 之间，用户可以自定义一些消息，通过 `SendMessage` 等函数传给窗口过程做自定义的处理工作，这时可以使用的值是从 `0400h` 开始的，`WM_USER` 就定义为 `00000400h`，当程序员定义多个用户消息的时候，一般使用 `WM_USER+1`, `WM_USER+2`, ... 之类的定义方法。

`wParam` 和 `lParam` 参数是消息所附带的参数，它随消息的不同而不同，对于不同的消息，它们的含义必须分别从手册中查明：如 `WM_MOUSEMOVE` 消息中，`wParam` 是标志，`lParam` 是鼠标位置；而在 `WM_GETTEXT` 消息中，`wParam` 是要获取的字符数，`lParam` 是缓冲区地址；而对于 `WM_COPY` 消息来说，它不需要额外的信息，所以两个参数都没有定义。

处理了不同的消息，必须返回规定的值给 Windows，返回值也需要分别从手册中查明，比如，处理 `WM_CREATE` 消息的时候，返回 0 表示成功；如果程序无法初始化，如申请内存失败，那么可以返回 -1，Windows 就不会继续窗口的创建过程。一些消息的返回值则没有定义，但大部分的消息处理以后都以返回 0 表示成功，所以程序中把默认的回语句放在最后，将 `eax` 清 0 后返回，如果在处理某个消息的时候需要返回不同的值，可以在分支中将 `eax` 赋值后直接用 `ret` 指令返回。对于 `DefWindowProc` 的返回值，我们不对它进行干涉，所以直接将 `eax` 不做修改地用 `ret` 返回。

`WM_CLOSE` 消息是按下了窗口右上角的“关闭”按钮后收到的，程序可以在这里处理和关闭窗口相关的事情，一般是相关资源的释放工作，如释放内存、保存工作和提示用户是否保存工作等，如记事本程序在未保存的时候单击“关闭”按钮，会有提示框提示是否先保存文件，单击“取消”按钮的话，记事本不会关闭，这个步骤就是在 `WM_CLOSE` 消息处理中完成的。如果处理 `WM_CLOSE` 消息时直接返回，那么窗口不会关闭，因为这个消息只是 Windows 通知窗口用户单击了“关闭”按钮而已，窗口采取什么样的行为是窗口的事。当窗口决定关闭的时候，需要程序自己调用 `DestroyWindow` 来摧毁窗口，并用 `PostQuitMessage` 向消息循环发送 `WM_QUIT` 消息来退出消息循环。调用 `PostQuitMessage` 时的参数是退出码，就是 `GetMessage` 收到 `WM_QUIT` 后 `MSG` 结构 `wParam` 字段中的内容，在这里使用 `NULL`。



PostQuitMessage 是初学者容易遗漏的函数，如果没有这条语句，外观上窗口是被摧毁掉，从屏幕上消失了，但主程序中的消息循环却没有收到 WM_QUIT，结果还在那里打转。常有人调试的时候丢了这条语句，结果再一次编译的时候就收到错误：LINK fatal error LNK1104: cannot open file "xxx.exe"，这就表示 exe 文件仍然被使用中。

Windows 为什么不在窗口摧毁的时候自动发送一个 WM_QUIT 消息，而必须由用户程序自己通过 PostQuitMessage 函数发送呢？其实很好理解：因为屏幕上可能不止一个窗口，Windows 无法确定哪个窗口关闭代表着程序结束。试想一下，用户打开了一个输入参数的小窗口，单击“确定”按钮后关闭并回到主窗口，Windows 却不分三七二十一自动发送了一个 WM_QUIT，程序就会莫名其妙地退出了。

2. 收到消息的顺序

窗口过程收到消息是有一定顺序的，收到第一条消息并不是从消息循环开始以后，而是在 CreateWindowEx 中就开始了，显示和刷新窗口的函数 ShowWindow 和 UpdateWindow 也向窗口过程发送消息，这一点并不奇怪，因为 Windows 在 CreateWindowEx 前调用 RegisterClassEx 的时候就已经得到窗口过程的地址了。并且在建立窗口的过程中需要窗口过程的配合。表 4.6 和表 4.7 分别列出了调用 CreateWindowEx 和 ShowWindow 的时候窗口过程收到的消息。

表 4.6 调用 CreateWindowEx 时窗口过程收到的消息

消息发生	说 明
WM_GETMINMAXINFO	获取窗口大小，以便初始化
WM_NCCREATE	非客户区开始建立
WM_NCCALCSIZE	计算客户区大小
WM_CREATE	窗口建立

表 4.7 调用 ShowWindow 时窗口过程收到的消息

消息发生	说 明
WM_SHOWWINDOW	显示窗口
WM_WINDOWPOSCHANGING	窗口位置准备改变
WM_ACTIVATEAPP	窗口准备激活
WM_NCACTIVATE	激活状态改变
WM_GETTEXT	取窗口名称（显示标题栏用）
WM_ACTIVATE	窗口准备激活
WM_SETFOCUS	窗口获得焦点

续表

消息发生	说 明
WM_NCPAINT	需要绘画窗口边框
WM_ERASEBKGD	需要擦除背景

WM_WINDOWPOSCHANGED	窗口位置已经改变
WM_SIZE	窗口大小已经改变
WM_MOVE	窗口位置已经移动

然后程序执行 UpdateWindow，这个函数仅仅向窗口过程发送一条 WM_PAINT 消息，接着，主程序开始进入消息循环，Windows 根据各种因素给窗口过程发送相应的消息，一直到调用 DestroyWindow 为止。表 4.8 列出了 DestroyWindow 向窗口过程发送的消息。

表 4.8 调用 DestroyWindow 时窗口过程收到的消息

消息发生	说 明
WM_NCACTIVATE	窗口激活状态改变
WM_ACTIVATE	窗口准备非激活
WM_ACTIVATEAPP	窗口准备非激活
WM_KILLFOCUS	失去焦点
WM_DESTROY	窗口即将被摧毁
WM_NCDESTROY	窗口的非客户区及所有子窗口已经被摧毁

在所有这些阶段的消息中，大部分的消息都不需要程序自己关心，Windows 只是尽义务通知窗口过程而已，窗口过程转手就交给 DefWindowProc 去处理了。程序需要关心的消息有下面这些，可以根据需要选择使用：

- WM_CREATE——放置窗口初始化代码，如建立各种子窗口（状态栏和工具栏等）。
- WM_SIZE——放置位置安排的代码，因为建立的子窗口可能需要随窗口大小的改变而移动位置。
- WM_PAINT——如果需要自己绘制客户区，则在这里安排代码。
- WM_CLOSE——向用户确认是否退出，如果退出则摧毁窗口并发送 WM_QUIT 消息。
- WM_DESTROY——窗口摧毁，在这里放置释放资源等扫尾代码。

在例子程序中，我们处理了 WM_PAINT 消息来绘制客户区，功能就是在窗口的中间写上一行字：“Win32 Assembly, Simple and powerful !”。窗口过程先通过 BeginPaint 获取窗口客户区的“设备环境”句柄，然后通过 GetClientRect 获取客户区的大小，最后通过 DrawText 函数将字符串按照取得的屏幕大小居中写到“设备环境”中，也就是窗口上。如果不需要显示这个字符串，则连 WM_PAINT 消息也不用处理。

3. 消息的默认处理——DefWindowProc

Windows 预定义的消息范围是 0~03ffh，共预留了 1 024 个消息编号，查看一下头文件 Windows.inc，可以发现实际已定义的消息数目有几百个，这些消息中的大部分对于窗口的运行来说都是必需的，如果窗口过程要处理每一种消息，那么窗口过程中的 elseif 语句就会绵延数千行，但是窗口的行为就是由处理这些消息的方法来表现的，不处理又不行，怎么办呢？

实际上，大部分窗口的行为都是差不多的，这意味着如果要窗口过程处理全部的消息，

不同窗口的窗口过程代码应该是大同小异的，完全可以用一个通用的模块来以默认的方式处理消息，Win32 中的 DefWindowProc 函数实现的就是这个功能。

不要小看了这个 DefWindowProc，正是它用默认的方式处理了几百种消息，才使用户能用区区百来行代码写出一个全功能的窗口。也正是所有的窗口都用 DefWindowProc 默认处理程序自己不处理的消息，才使它们的行为看上去大同小异，因为它们背后实际上是同一块代码在处理。

在窗口过程的分支语句中，用户处理所有需要个性化处理的消息，对于表现为默认行为的消息，则在 else 分支中用 DefWindowProc 来处理。由于对于 Windows 来说，它并不关心消息在窗口过程中是程序用自己的代码处理的还是用 DefWindowProc 处理的，它只看 eax 中的返回值来了解处理结果，所以不管消息是谁处理的，都必须在 eax 中返回正确的值。DefWindowProc 返回时 eax 中就是它对消息的处理结果，程序只要直接把 eax 传回给 Windows 就行了，所以在例子程序中，DefWindowProc 后面直接用一句 ret 指令返回。



注意：例子中 DefWindowProc 后面直接使用的这句 ret 非常重要，如果丢失了这一句，那么相当于处理大多数消息时没有返回正确的值，窗口将不会正常工作。

表 4.9 中列出了 DefWindowProc 中对一些消息的处理方法，如果与用户期望的不同，就必须在窗口过程中自己处理。

表 4.9 DefWindowProc 对一些消息的默认处理方式

消 息	DefWindowProc 的处理方式
WM_PAINT	发送 WM_ERASEBKGND 消息来擦除背景
WM_ERASEBKGND	用窗口类结构中的 hbrBackground 刷子来绘制窗口背景
WM_CLOSE	调用 DestroyWindow 来摧毁窗口
WM_NCLBUTTONDBLCLK	这是非客户区（如标题栏）鼠标双击消息，DefWindowProc 测试鼠标的位置，然后再采取相应的措施，如标题栏双击将最大化和恢复窗口
WM_NCLBUTTONUP	这是非客户区鼠标释放消息，同样，DefWindowProc 测试鼠标的位置然后再采取相应的措施，如鼠标在“关闭”按钮的位置释放将导致发送 WM_CLOSE 消息
WM_NCPAINT	非客户区绘制消息，DefWindowProc 将绘制边框和客户区

从这些默认的处理方法可以看出，想要一个窗口和别的窗口看起来不一样，比如，想要窗口看起来像苹果机的窗口一样，并且把关闭按钮移到标题栏最左边去，那么可以自己处理 WM_NCPAINT 消息，把非客户区画成苹果机窗口的样子，并把关闭按钮画到标题栏左边去，并且自己处理 WM_NCLBUTTONUP 消息，当检测到鼠标按下的位置在自己的关闭按钮上的时候，则发送 WM_CLOSE 消息。对别的消息的处理思路也可以按这种方法类推。

另外，可以发现 DefWindowProc 对 WM_CLOSE 的默认处理是调用 DestroyWindow 摧毁窗口，DestroyWindow 会引发一个 WM_DESTROY 消息，WM_CLOSE 和 WM_DESTROY 的不同之处是：WM_CLOSE 代表用户有关闭窗口的意向，窗口过程有权不“服从”，但收到 WM_DESTROY 的时候窗口已经在关闭过程中了，不管窗口过程愿不愿意，窗口的关闭已经是不可挽回的

事了。

对于这两个消息，窗口过程必须处理其中的一个，因为必须有个地方发送 WM_QUIT 消息来结束消息循环，例子程序中处理 WM_CLOSE 消息，在其中用 DestroyWindow 摧毁窗口，再调用 PostQuitMessage 结束消息循环；程序也可以不处理 WM_CLOSE 消息，让 DefWindowProc 以默认处理的方式摧毁窗口，但这时候必须处理 WM_DESTROY 消息，在其中调用 PostQuitMessage 发送 WM_QUIT 以结束消息循环。

附录 B（以电子版方式放在随书光盘中）中的内容以几个实验的方式，演示了窗口从建立、进行各种操作到摧毁的过程中收到的各种消息，读者可以自行阅读并进行实验，以加深对窗口程序工作原理的认识。

4.3 窗口间的通信

4.3.1 窗口间的消息互发

在介绍消息循环的时候，已经知道在不同应用程序之间的窗口中可以互发消息（如图 4.4 所示），方法是通过 SendMessage 或者 PostMessage 函数，这两个函数的使用语法是相同的：

invoke	PostMessage, hWnd, Msg, wParam, lParam
invoke	SendMessage, hWnd, Msg, wParam, lParam.

对于不同的 Msg, wParam 和 lParam 的含义是不同的，如对于 WM_SETTEXT 是：

wParam = 0;	;未定义，必须为 0
lParam = (LPARAM) (LPCTSTR) lpsz	;要设置的字符串地址

想一想就会发现一个问题：Windows 中不同应用程序的地址空间是隔离的（如图 1.6 所示），假设程序 1 要用 SendMessage 调用程序 2 所属窗口的窗口过程，但程序 2 窗口过程的代码并不在程序 1 的地址空间中，那么 SendMessage 如何调用它呢？其实很简单，当程序 1 调用 SendMessage 函数的时候，Windows 会先保存 wParam 和 lParam 参数并等待，等轮到程序 2 的时间片的时候再去调用它的窗口过程，并把保存的 wParam 和 lParam 参数发给它，等窗口过程返回的时候，Windows 记下返回值并等待，再等轮到程序 1 的时间片的时候把返回值当做 SendMessage 的返回值传给程序 1，这样程序 1 看上去就像自己直接在调用程序 2 的窗口过程一样。

但又一个问题出现了：Windows 在做“牵线红娘”的时候传递了 wParam 和 lParam，以及返回值，如果参数指向一个字符串呢，比如说上面的 WM_SETTEXT 消息中的 lParam 指向一个字符串，假设程序 1 中 lParam 指向字符串的地址为 xxxxxxxx，把这个地址传给程序 2 的时候，程序 2 不可能访问到程序 1 的地址空间，在程序 2 中 xxxxxxxx 指向的可能是其他内容，也可能是不可访问的，这又该如何处理呢？

写一个源程序实验一下，用一个程序向另一个程序的窗口发送 WM_SETTEXT 消息，然

哭即

.elseif	eax ==	WM_SETTEXT
	invoke	wsprintf, addr szBuffer, addr szReceive, lParam, lParam
	invoke	MessageBox, hWnd, offset szBuffer, addr szCaptionMain, MB_OK

同时在数据段中加上下列定义：

szCaptionMain	db	'Receive Message', 0
szReceive	db	'Receive WM_SETTEXT message', 0dh, 0ah
	db	'param: %08x', 0dh, 0ah
	db	'text: "%s"', 0dh, 0ah, 0

```
int wprintf(  
    LPTSTR lpOut,        // 输出缓冲区地址  
    LPCTSTR lpFmt,        // 格式化串地址  
    ...                  // 变量列表  
);
```

invoke	wsprintf, addr szBuffer, addr szReceive, lParam, lParam
--------	---

invoke	MessageBox, hWnd, offset szBuffer, addr szCaptionMain, MB_OK
--------	--

接收程序写好了，现在来写一个发送程序，如下所示：

```

        .386
        .model flat,stdcall
        option casemap:none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include    windows.inc
include    user32.inc
includelib user32.lib

```

```

include      kernel32.inc
includelib   kernel32.lib
;>>>>>>>>>
        .data
hWnd       dd      ?
szBuffer    db      256 dup (?)
;>>>>>>>>>
        .const
szCaption db      'SendMessage',0
szStart     db      'Press OK to start SendMessage, param: %08x!',0
szReturn     db      'SendMessage returned!',0
szDestClass db      'MyClass',0
szText       db      'Text send to other windows',0
szNotFound   db      'Receive Message Window not found!',0
;>>>>>>>>>
        .code
start:
        invoke    FindWindow,addr szDestClass,NULL
        .if      eax
                mov     hWnd,eax
                invoke  sprintf,addr szBuffer,addr szStart,addr szText
                invoke  MessageBox,NULL,offset szBuffer,\
                        offset szCaption,MB_OK
                invoke  SendMessage,hWnd,WM_SETTEXT,0,addr szText
                invoke  MessageBox,NULL,offset szReturn,\
                        offset szCaption,MB_OK
        .else
                invoke  MessageBox,NULL,offset szNotFound,\
                        offset szCaption,MB_OK
        .endif
        invoke    ExitProcess,NULL
;>>>>>>>>>
        end      start

```

在这个程序中首先用 FindWindow 函数找到接收窗口的窗口句柄，FindWindow 函数的使用方法是：

```

invoke    FindWindow,lpClassName,lpWindowName
.if      eax
        mov     hWin,eax
.endif

```

两个参数都指向字符串，lpClassName 指向需要寻找的窗口的窗口类，lpWindowName 指向需要寻找窗口的窗口标题，如果目标窗口存在的话，函数的返回值是找到的窗口句柄，否则函数返回 0。

用接收窗口的窗口类当做参数寻找窗口，如果没有找到，显示“Receive Message Window not found”，找到的话则把“Text send to other windows”字符串的地址当做 WM_SETTEXT 消息的参数用 SendMessage 发送给接收窗口。两个程序的源代码可以在所附带光盘的 Chapter04\SendMessage 目录中找到。

好！现在发送开始，首先执行 Receive.exe，窗口出来了，然后执行 Send.exe，屏幕

上出现一个对话框：Press OK to start SendMessage, param: 00402072，表示在 Send 程序中字符串的地址是 00402072h，现在单击“确定”按钮执行 SendMessage 函数，单击后对话框消失，但接收程序显示出了一个对话框，内容为：

```
Receive WM_SETTEXT message
param: 0012ff1c                      (注：该地址在具体执行的时候可能有所不同)
text: "Text send to other windows"
```

可见字符串是正确地传了过来，但地址却不是发送程序的 00402072h，这是怎么回事呢？

其实 Windows 在处理 SendMessage 的时候要检查消息的类型，并对不同的消息做不同的处理，当消息的参数是一个普通的 32 位数时，仅仅将该数值传递给目标窗口过程；而当消息的参数是一个指针的时候，Windows 对指针指向的内容进行了一些处理，以便数据能够正常地传递到目标进程中。

Windows 首先创建一块共享内存，并将 WM_SETTEXT 消息 lParam 指向的字符串拷贝到该内存中，然后再发送消息到其他进程，并将共享内存存在目标进程中的地址发送给目标窗口过程，目标窗口过程处理完消息后，函数返回，共享内存被释放。共享内存使用的是第 10 章介绍的内存映射文件技术，相当于用图 1.6 的方法将同一块物理内存映射到不同进程的不同线性地址上去。

虽然当消息传递到目标窗口过程的时候 lParam 的取值会有所变化，但在 WM_SETTEXT 消息中，lParam 的数值是多少并不重要，重要的是它指向的字符串是否正确。

最后，单击 Receive 程序中的“确定”按钮，Send 程序马上会弹出一个消息框并显示：SendMessage returned，这是 SendMessage 函数告诉我们：我回来了！



在用户自定义的消息中（WM_USER 等），不要在消息参数中传递指针，这只会引发非法访问内存，因为 Windows 不知道用户的意图，它只会把 lParam 和 wParam 当两个普通的数值传递，而不会帮用户把指针指向的内容复制到一块共享内存中。

4.3.2 在窗口间传递数据

在 WM_SETTEXT 这一类的消息中，Windows 可以将参数所指的字符串传递到目标窗口过程中，但是这些消息都有它们的本职工作，并且传递的数据也只限于以 0 结尾的字符串。为了能够在不同进程的窗口间自由地拷贝任意类型的数据，Windows 提供了一个特殊的窗口消息——WM_COPYDATA。

WM_COPYDATA 消息用一个 COPYDATASTRUCT 结构来描述要拷贝的数据的长度和位置：

```
COPYDATASTRUCT STRUCT
    dwData  DWORD    ?           ; 附加字段
    cbData   DWORD    ?           ; 数据长度
    lpData   DWORD    ?           ; 数据位置指针
COPYDATASTRUCT ENDS
```

其中的 dwData 字段是一个备用的字段，可以存放任何值，例如，读者有可能向另外

的进程发送数据的同时用一个数字来说明数据的类型，那么就可以把这个字段用上去；cbData 字段规定了发送的字节数，lpData 字段是指向待发送数据的指针。填充好数据结构后，用 SendMessage 函数就可以将数据发送给目标窗口过程：

```
.data
stCopyData    COPYDATASTRUCT <>
.code
...
invoke    SendMessage, hDestWnd, WM_COPYDATA, hWnd, addr stCopyData
```

例句中的 hDestWnd 为目标窗口句柄；wParam 指定为 hWnd，是当前窗口的句柄；lParam 指向已经填充完毕的 COPYDATASTRUCT 结构。

Windows 收到 WM_COPYDATA 消息后，会根据 cbData 字段的长度创建一块共享内存，并把 lpData 所指的数据拷贝到共享内存中，然后定位该共享内存存在目标进程中的地址，把该地址作为新的地址添加到 COPYDATASTRUCT 结构的 lpData 字段中，最后将经过处理的 COPYDATASTRUCT 结构发送给目标窗口过程，目标窗口过程就可以根据结构中的字段来定位数据了。目标窗口过程返回后，Windows 释放掉共享内存，SendMessage 函数返回。

光盘 Chapter04\SendMessage-1 目录中的源代码演示了 WM_COPYDATA 消息的使用方法，读者可自行对比该例子和上一个例子的区别。

4.3.3 SendMessage 和 PostMessage 函数的区别

从逻辑上看，SendMessage 函数相当于直接调用其他窗口的窗口过程来处理某个消息，并等待窗口过程的返回，在函数返回后，目标窗口过程必定已经处理了该消息。PostMessage 函数则将消息放入目标窗口的消息队列中并直接返回，函数返回后，目标窗口过程可能还没有处理到该消息。

对于普通的消息来说，两个函数除了在处理速度上有所区别外，其他的表现都一模一样，但是对于 WM_SETTEXT，WM_COPYDATA 等在参数中用到指针的消息来说，两者就有所不同了。读者可以尝试将前面例子中的 SendMessage 函数改为 PostMessage 函数，就会发现 Receive 程序根本不会接收到 WM_SETTEXT 或者 WM_COPYDATA 消息。事实上，当消息参数中用到指针时，用 PostMessage 函数来发送消息都不会成功，PostMessage 的参考文档中明确地说明，该函数不能用于任何参数中用到指针的消息。

第 5 章

使用资源

读者可能注意到大多数 Windows 程序都包含图标。打开“我的电脑”以后，各个可执行程序显示的图标各不相同，当程序运行后，大多数程序有菜单。另外，当鼠标移动到窗口中后，光标也有可能变得不同，很多程序还使用对话框来提供用户界面。

菜单、图标与对话框都是可执行文件的组成部分，它们是以资源的形式存放在文件中的。但这些资源并不在源代码的数据段中定义，而是由链接程序放入文件的单独一个节区中，当运行中要用到资源的时候，必须借助 API 函数装入后才能使用。

除了菜单、图标与对话框，Windows 中还有其他一些类型的资源，它们是：

- 菜单和加速键
- 光标和图标
- 位图
- 对话框
- 字符串资源
- 版本信息
- 自定义资源

资源文件的“源文件”是以 rc 为扩展名的脚本文件，由资源编译器 Rc.exe 编译成为以 res 为扩展名的二进制资源文件，最后在链接的时候由 Link.exe 链入可执行文件中，这在前面的内容中已经有所介绍，在这一章中将介绍资源的定义方法，以及在程序中的使用方法。

本章的篇幅比较大，但是编写一个 Win32 程序，与界面有关的代码起码要占一半以上，而与界面相关的代码中，又有大部分涉及各种资源和控件的使用，所以仔细研究本章，以及第 9 章介绍的“通用控件”的内容绝不是浪费时间，了解了这两章的内容，写一个应用程序的界面就基本上不成问题了。

5.1 菜单和加速键

5.1.1 菜单和加速键的组成

如图 5.1 所示，在窗口中，菜单位于标题栏下面。这个菜单称为“主菜单”或“顶层菜单”，图中菜单的菜单项有“文件”、“查看”和“帮助”。单击主菜单上的项目后，可以弹出下一层菜单，叫做“弹出式菜单”或“子菜单”。子菜单中可以继续包含下一层子菜单。如单击“查看”弹出一个子菜单后，再单击其中的“工具栏”可以继续弹出一个子菜单。在子菜单中可以继续弹出下一层子菜单的菜单项最右边用一个三角箭头来表示。

有的程序在窗口的客户区单击鼠标右键也可以弹出一个菜单，单击标题栏图标也可以弹出一个系统菜单，这些菜单都属于弹出式菜单。

菜单中的菜单项有好几种，从资源定义的角度来看，分隔用的横线也是一个菜单项。除横线外其他菜单项可以供用户选择，也可以设置为“禁止”或“灰化”状态暂时停用，如图 5.1 中“被禁用的菜单项”和“被灰化的菜单项”所示。“禁用”的菜单项看上去和普通菜单项相同，但无法在上面单击鼠标，“灰化”的菜单项从外观上就已经表示是不可用的。菜单项也可以在左边显示选中标记，如图 5.1 中的“大图标”前的圆点和“状态栏”前的打钩。圆点表示选中标记是互斥的，打钩表示是不互斥的。

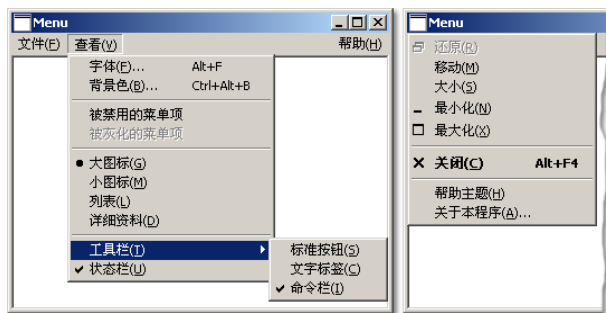


图 5.1 菜单示意图

加速键就是菜单项的快捷键，图中的“字体”菜单项右边有个“Alt+F”，表示当窗口是激活的时候，不必打开菜单，直接按“Alt”加“F”的组合键就相当于选择了“字体”菜单项，同样，直接按下“Ctrl”加“Alt”加“B”键等于选择了“背景色”菜单项。加速键也是资源的组成部分，一般将最常用的菜单项定义为加速键，以减少打开菜单的操作。加速键的定义要遵循惯例，如“Ctrl+C”和“Ctrl+V”一般定义为“拷贝”和“粘贴”，“Ctrl+X”定义为“剪切”等。当然加速键的定义并不是必需的，不定义加速键并不会影响程序的功能。

不管程序中是否定义加速键，Windows 总是定义了几个默认的加速键，如“F10”键会打开窗口的主菜单，“Alt+空格”会打开系统菜单，“Alt+F4”等于单击了“关闭”按钮等。

哭

哭

哭

编译上述文件使用的 makefile 文件如下:

为了编译资源文件，makefile 中比以前多了一个资源编译的隐含规则：

同时，在 exe 文件的依赖文件中增加了 Menu.res 文件。

122

法在后面加 h，而是用前面加 0x 的方法，如 1234h 写为 0x1234，注释也要用前面加 // 的方法。这些在书写的时候一定要注意，以免引起语法错误。

在脚本文件的头部，首先要把 MASM32 SDK 软件包中的 resource.h 文件包含进来，这个文件中包括了资源定义中很多的预定义值，如窗口属性与加速键的键值等。资源在程序中的引用往往用一个数值来表示，称为资源的 ID 值，但在定义的时候直接使用数值不是很直观，所以往往用#define 语句将数值定义为容易记忆的字符串。

1. 菜单的定义

在资源脚本文件中菜单的定义格式是：

```
菜单 ID MENU [DISCARDABLE]
BEGIN
    菜单项定义
    ...
END
```

“菜单 ID MENU [DISCARDABLE]”语句用来指定菜单的 ID 值和内存属性，菜单 ID 可以是 16 位的整数，范围是 1~65 535，在 Menu.rc 文件中，定义的菜单 ID 是 2000h，但菜单 ID 也可以用字符串表示，如下面的定义：

```
MainMenu menu
begin
    menuitem ...
end
```

表示菜单的 ID 是字符串型的“MainMenu”，但这样定义的话，在程序中引用的时候就要用字符串指针代替十六进制的菜单 ID 值，显得相当不便，所以在实际应用中通常使用十六进制数值当做菜单 ID。

数值型 ID 的范围限制在 1~65 535 之间的原因是字符串在内存中的线性地址总是大于 10000h，API 函数检测参数时发现小于 10000h 时就可以把它认为是数值型的，大于 10000h 时就当做字符串指针处理。

menu 关键字后面的 DISCARDABLE 是菜单的内存属性，表示菜单在不再使用的时候可以暂时从内存中释放以节省内存，这是一个可选属性。菜单项的定义语句必须包含在 begin 和 end 关键字之内，这两个关键字也可以用花括号{ 和 } 代替。

菜单项目的定义方法有 3 类：

```
MENUITEM 菜单文字, 命令 ID [, 选项列表]    (用法 1)
或 MENUITEM SEPARATOR                      (用法 2)
或 POPUP 菜单文字 [, 选项列表]              (用法 3)
BEGIN
    item-definitions
    ...
END
```

下面分别就这 3 类详细说明，用法 1 定义的是普通菜单项，图 5.1 中的“字体”与“背景色”等菜单项都是这样定义的，它的组成部分如下：

- 菜单文字——显示在菜单项中的字符串。如果需要字符串中某个字母带下划线，那么可以在字母前面加&符号，如“字体(F)...”就要写成“字体(&F)...”，带下划线的字母可以被系统自动当做快捷键：在这里，当菜单打开的时候按下 F 键，那么就相当于用鼠标选择了“字体”选项。在同一个弹出菜单中要注意不同的菜单项快捷键应该有所区别。另外，如果要把加速键的提示信息显示在菜单项的右边，如“字体”菜单项中的“Alt+F”字符，可以在两者中间加\t（表示插入一个 Tab 字符），写为“字体(&F)...\tAlt+F”，这样 Tab 后面的字符在显示的时候会右对齐。
- 命令 ID——用来分辨不同的菜单项。当菜单被选中的时候，Windows 会向窗口过程发送 WM_COMMAND 消息，消息的参数就是这个命令 ID。用命令 ID 可以分辨用户究竟选中了哪个菜单项，所以不同的菜单项应该定义不同的 ID 值，除非想让两个菜单项的功能相同。
- 选项——用来定义菜单项的各种属性，它可以是下列数值：
 - CHECKED——表示打上选定标志（对钩）。
 - GRAYED——表示菜单项是灰化的。
 - INACTIVE——表示菜单项是禁用的。
 - MENUBREAK 或 MENUBARBREAK——表示将这个菜单项和以后的菜单项列到新的列中。

读者可以做个实验，把例子中的“详细资料”一句的定义语句改为：

menuitem	"详细资料(&D)", IDM_DETAIL, MENUBARBREAK
----------	--------------------------------------

那么显示出来的菜单如图 5.2 所示：“详细资料”及以后的菜单项都另起一列了！

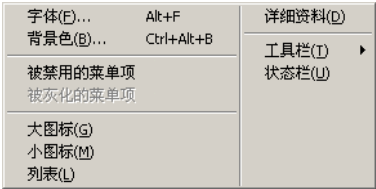


图 5.2 使用 MENUBARBREAK 的效果

用法 2 定义的是菜单项之间的分隔线，显然，分隔线是不需要字符串和选项的。

方法 3 定义的是弹出式菜单，顶层菜单是由多个弹出式子菜单组成的，所以在 Menu.rc 文件中，主菜单是由“文件”、“查看”和“帮助”3 个顺序定义的弹出式菜单组成的，弹出式菜单的定义也可以嵌套，如“查看”菜单中的“工具栏”又是一个弹出式菜单，在嵌套的时候要注意像写 C 的源程序一样把 begin 和 end（或者 { 和 }）正确地配对。popup 菜单的选项列表可以是以下的值：

- GRAYED——表示菜单项是灰化的。
- INACTIVE——表示菜单项是禁用的。

● HELP——表示本项和以后的菜单项是右对齐的，如图 5.1 中所示的“帮助”菜单。

由于 popup 菜单项选中的时候会自动将弹出式菜单弹出来，不需要向程序发送消息，所以在定义的参数中不需要命令 ID。

有些选项是可以同时定义的，如果要指定超过一个的选项，中间要用逗号隔开，但是也有些小小的限制：GRAYED 和 INACTIVE 不能同时使用，MENUBREAK 和 MENUBARBREAK 也是不能同时使用的。

2. 加速键的定义

与菜单的定义相比，加速键的定义要简单得多，具体的语法如下：

```
加速键 ID ACCELERATORS
BEGIN
    键名, 命令 ID [, 类型] [, 选项]
    ...
END
```

加速键 ID 同样可以是一个字符串或者是 1~65 535 之间的数字，整个定义内容也是用 begin 和 end（或花括号）包含起来，中间是多个加速键的定义项目，每个键占据一行，各字段的含义如下所示。

- 键名——表示加速键对应的按键，可以有 3 种方式定义。
 - “^字母”：表示 Ctrl 键加上字母键。
 - “字母”：表示字母，这时类型必须指明是 VIRTKEY。
 - 数值：表示 ASCII 码为该数值的字母，这时类型必须指明为 ASCII。
- 命令 ID——按下加速键后，Windows 向程序发送的命令 ID。如果想把加速键和菜单项关联起来，这里就是要关联菜单项的命令 ID。
- 类型——用来指定键的定义方式，可以是 VIRTKEY 和 ASCII，分别用来表示“键名”字段定义的是虚拟键还是 ASCII 码。
- 选项——可以是 Alt, Control 或 Shift 中的单个或多个，如果指定多个，则中间用逗号隔开，表示加速键是按键加上这些控制键的组合键。

在键名的定义中，系统按键如 F1, F2, BackSpace 和 Esc 等都是用虚拟键的方法定义的，Resource.h 中已经包括所有的预定义，它们是以 VK_ 带头的一些值，如 VK_BACK, VK_TAB, VK_RETURN, VK_ESCAPE, VK_DELETE, VK_F1 和 VK_F2 等，读者可以查看 Resource.h 文件。下面是加速键定义的一些例子：

"^C",	ID	;Ctrl+C
"K",	ID	;Shift+K
"k",	ID, ALT	;Alt+k
98,	ID, ASCII	;b (字符 b 的 ASCII 码为 98)
66,	ID, ASCII	;B (Shift b)
"g",	ID	;g
VK_F1,	ID, VIRTKEY	;F1

(1) 对于同类别的多个资源，资源 ID 必须为不同的值，如定义了两个菜单，那么它们的 ID 就必须用不同的数值表示，否则将无法分辨。

5.1.3 使用菜单和加速键

- 程序在用户选择了任何一个菜单项以后，会弹出一个对话框，将接收到的菜单命令 ID 显示出来。
- 选择“大图标”、“小图标”、“列表”和“详细资料”菜单项后，选中的菜单项前面会出现一个圆点选中标记，4 个菜单项的选择是互斥的。
- 在“状态栏”及“工具栏”菜单的 3 个菜单项中选择后，选中的菜单项前面会出现打钩标记，它们是不互斥的。
- 在窗口的客户区单击鼠标右键会弹出和“查看”菜单一致的弹出式菜单。
- 在标题栏图标上单击鼠标左键，会弹出系统菜单，注意上面比默认的菜单多了两项：“帮助主题”和“关于本程序”。

[illegible]

[illegible]

128

哭

1. 加载菜单

invoke	CreateWindowEx, WS_EX_CLIENTEDGE, \ offset szClassName, offset szCaptionMain, \ WS_OVERLAPPEDWINDOW, \ 100, 100, 400, 300, \ NULL, hMenu , hInstance, NULL
--------	---

invoke	LoadMenu , hInstance, IDM_MAIN
mov	hMenu, eax

当资源文件中用字符串为名称定义菜单而不是用数值的时候，例如：

```

MainMenu      menu      //定义菜单名为字符串“MainMenu”
begin
    ...
end

```

那么在程序中就必须用字符串指针代替菜单 ID 做参数：

```

szMenu        "MainMenu", 0 ;在数据段中定义菜单名称字符串
    ...
    invoke      LoadMenu, hInstance, addr szMenu ;在程序中装载
    mov         hMenu, eax

```



用字符串为名称定义资源，在 LoadMenu、LoadCursor、LoadBitmap 等资源装载函数中用字符串指针做参数装入，这实际上是一个通用的方法，不仅适用于菜单资源，对于其他类别的资源也是适用的。在后面的介绍中就不再另外说明了。

2. 加载加速键

和菜单一样，加速键在使用前也要装入，参数同样是在资源脚本文件中定义的加速键 ID，程序中对应的语句是：

```

invoke      LoadAccelerators, hInstance, IDA_MAIN
mov         @hAccelerator, eax

```

其实，我们自己在程序中也可以很方便地实现加速键功能，方法是在 WM_KEYDOWN 消息中判断键盘消息并按照自定义的逻辑进行处理，使用加速键实际上是让 Windows 替我们完成这个功能，Windows 实现的方法正是在消息循环中检查 WM_KEYDOWN 和 WM_SYSKEYDOWN 消息。下面是使用加速键时消息循环的代码，请注意粗体字部分：

```

.while      TRUE
    invoke   GetMessage, addr @stMsg, NULL, 0, 0
    .break  .if eax == 0
    invoke   TranslateAccelerator, hWinMain, @hAccelerator, addr @stMsg
    .if      eax == 0
        invoke   TranslateMessage, addr @stMsg
        invoke   DispatchMessage, addr @stMsg
    .endif
.endif
.endw

```

TranslateAccelerator 函数是实现加速键功能的核心，它的参数为目标窗口、加速键句柄和 GetMessage 取得的消息结构。该函数检查消息结构中的消息，如果遇到 WM_KEYDOWN 和 WM_SYSKEYDOWN 消息则检测加速键资源，看按键是否符合某个加速键，符合的话则向目标窗口发送 WM_COMMAND 或 WM_SYSCOMMAND 消息，并返回 TRUE，不符合则不进行任何处理并返回 FALSE。

由于加速键的键码并不是用户真正想输入窗口的，比如，用户在写字板中输入文字，按 Ctrl+C 键是为了“拷贝”，而并不是想把 Ctrl+C 键对应的字符输入文档，所以这个 Ctrl+C 的键码在完成加速键的使命后就应该丢弃，也就是说，符合加速键的键盘消息不

应该再发送给窗口, TranslateMessage 和 DispatchMessage 函数前的逻辑判断就是这样的意图: 只有 TranslateAccelerator 没有转换的消息 (返回值 eax 为 0) 才继续处理。

另外, TranslateAccelerator 的参数中有个“目标窗口”, 例子中是程序的主窗口 hWinMain, 为什么要设置这样一个参数而不像 DispatchMessage 函数一样使用 MSG 结构中的 hwnd 呢? 这是因为键盘消息可以产生于不同窗口中——既可能是主窗口, 也可能是其他子窗口, 如果用 @stMsg.hwnd 做目标窗口, 就必须在所有子窗口的窗口过程中都设置处理加速键消息的代码, 这显然是一种浪费, 所以一般把所有的加速键消息都发送到主窗口, 然后集中在主窗口的窗口过程中处理 WM_COMMAND 消息, 这样有利于精简代码。

3. 菜单和加速键消息

当用户选择了一个菜单项的时候, Windows 向菜单所属的窗口发送 WM_COMMAND 消息; 而用户按下了一个加速键的时候, Windows 向 TranslateAccelerator 函数指定的目标窗口发送 WM_COMMAND 消息。一般这两种情况对应的窗口都是主窗口, 所以可以在主窗口中的窗口过程中集中处理 WM_COMMAND 消息, 而不必考虑它究竟是菜单引发的还是加速键引发的。

WM_COMMAND 消息的两个参数是这样定义的:

wParam 的高位	=	wNotifyCode	;通知码
wParam 的低位	=	wID	;命令 ID
lParam	=	hwndCtl	;发送 WM_COMMAND 的子窗口句柄

除了菜单和加速键, WM_COMMAND 消息也可以由其他子窗口引发, 如主窗口中的按钮或工具栏等, lParam 参数指定了引发消息的子窗口句柄, 对于菜单和加速键引发的 WM_COMMAND 消息, lParam 的值为零。wParam 参数的低 16 位是命令 ID, 也就是资源脚本文件中菜单项的命令 ID 或加速键的命令 ID, 高 16 位是通知码, 菜单消息的通知码是 0, 加速键消息的通知码为 1。

在需要处理菜单和加速键消息的窗口过程中, 一般需要增加一个 WM_COMMAND 分支来处理对应的消息, 这个分支的一般结构为:

```

.elseif    eax ==    WM_COMMAND        ;eax 中为 wParam
    mov     eax, wParam
    movzx   eax, ax
    .if     eax ==    命令 ID1
        ...
    .elseif eax ==    命令 ID2
        ...
    .endif

```

其中 movzx eax, ax 指令将 16 位的 ax 扩展到 32 位的 eax, 相当于将 eax 的高 16 位清零, 作用就是当消息由加速键引起时, 将高 16 位中的 1 忽略, 这样下面的分支就可以同时处理菜单和加速键消息, 当然读者也可以去掉这一句, 这时下面的比较语句中就要使用 ax 而不是 eax。

在例子程序中, mov eax, wParam 前面还有一句 invoke _DisplayMenuItem, wParam,

作用是在处理 WM_COMMAND 消息前将 wParam 的值通过一个对话框显示出来，读者可以与资源脚本文件中定义的命令 ID 值对比一下，在正常使用的程序中可以去掉这一句。

读者可以发现，资源文件中定义的“字体”菜单项的 ID 为 0x4201，当选中“字体”菜单项的时候，对话框中显示的 wParam 数值正是 00004201，而按下加速键 Alt+F 的时候，显示出来的值就是 00014201 了，它们的区别就是高 16 位中的通知码不同。

4. 菜单项的修改

在程序的运行中也可以动态修改菜单项，包括添加、删除和修改操作，这些操作是通过几个 API 函数来完成的：

invoke	AppendMenu, hMenu, uFlags, uIDNewItem, lpNewItem	;添加菜单项
invoke	InsertMenu, hMenu, uPosition, uFlags, uIDNewItem, lpNewItem	;插入菜单项
invoke	ModifyMenu, hMenu, uPosition, uFlags, uIDNewItem, lpNewItem	;修改菜单项
invoke	DeleteMenu, hMenu, uPosition, uFlags	;删除菜单项
invoke	RemoveMenu, hMenu, uPosition, uFlags	;删除菜单项

其中 AppendMenu 用来在一个菜单的最后添加菜单项，InsertMenu 则在中间插入菜单项，ModifyMenu 可以修改一个菜单项的文字，DeleteMenu 和 RemoveMenu 则可以删除一个菜单项。

这些函数中的参数都是雷同的，hMenu 参数指要操作的菜单句柄；uPosition 用来定位要操作的菜单项。定位的方法有两种：用命令 ID 定位或用位置索引。用哪一种方法取决于后面的 uFlags 参数：当 uFlags 为 MF_BYCOMMAND 时，uPosition 为菜单项的命令 ID；而当 uFlags 为 MF_BYPOSITION 的时候，uPosition 表示菜单项的位置索引，索引是从 0 开始的，也就是说第一个菜单项的索引值为 0。

由于 AppendMenu 函数总是在菜单的最后添加新的菜单项，所以不需要 uPosition 参数。

对于 AppendMenu 和 InsertMenu，会有一个新的菜单项产生，uIDNewItem 表示这个新菜单项的命令 ID，lpNewItem 指向新菜单项的文字字符串，ModifyMenu 函数可以修改一个菜单项的命令 ID 或文字字符串，所以也有 uIDNewItem 和 lpNewItem 参数。而用来删除菜单项的 DeleteMenu 和 RemoveMenu 显然用不着 uIDNewItem 和 lpNewItem 参数。

uFlags 参数除了指定 MF_BYCOMMAND 还是 MF_BYPOSITION 外，还可以组合指定菜单项的其他属性，如 MF_CHECKED, MF_DISABLED, MF_ENABLED, MF_GRAYED, MF_MENUBARBREAK, MF_MENUBREAK, MF_SEPARATOR 和 MF_UNCHECKED 等，从其字面上就可以看出这些属性的含义。

DeleteMenu 和 RemoveMenu 的不同之处在于对 popup 菜单项的处理。当它们用于 popup 属性的菜单项时，DeleteMenu 不仅删除菜单项，而且将这个 popup 菜单项的所有子项目全部删除，这样，这个 popup 菜单就不能在别的地方继续使用；而 RemoveMenu 仅从菜单中移去这个 popup 菜单项，整个 popup 菜单在内存中还是存在的。以 Menu.asm 程序为例，按鼠标右键弹出的菜单实际上是主菜单中的“查看”菜单项，假如用 DeleteMenu 删除主菜单中的“查看”项目，那么按右键也就弹不出菜单了，而用 RemoveMenu 删除主菜单中

的“查看”项目，按鼠标右键仍然可以弹出菜单。对于非 popup 属性的菜单项，DeleteMenu 和 RemoveMenu 的效果是相同的。

5. 使用系统菜单

系统菜单指按下了标题栏图标后弹出的菜单，与窗口菜单不同，选中系统菜单的菜单项后，Windows 向窗口发送的是 WM_SYSCOMMAND 消息而非 WM_COMMAND 消息。默认的系统菜单中已经有“还原”、“移动”、“大小”、“最大化”、“最小化”和“关闭”等菜单项，这些菜单项的命令 ID 已经预定义为 SC_RESTORE, SC_MOVE, SC_SIZE, SC_MAXIMIZE, SC_MINIMIZE 和 SC_CLOSE 等，如果读者要自己处理它们，可以在 WM_SYSCOMMAND 消息中建立一个比较分支对它们进行处理，一般在程序中并不自己处理 WM_SYSCOMMAND 消息，而是交给 DefWindowProc 处理。

如何在系统菜单中添加自己的菜单项呢？方法就是使用上面介绍的 AppendMenu（当然也可以用 InsertMenu），在添加前必须用 GetSystemMenu 函数首先获取系统菜单的句柄。例子程序在窗口初始化的时候在系统菜单尾添加了一个分隔线和两个菜单项：“帮助主题”和“关于本程序”：

```
.if      eax ==    WM_CREATE
...
invoke   GetSystemMenu, hWnd, FALSE
mov      @hSysMenu, eax
invoke   AppendMenu, @hSysMenu, MF_SEPARATOR, 0, NULL
invoke   AppendMenu, @hSysMenu, 0, IDM_HELP, offset szMenuHelp
invoke   AppendMenu, @hSysMenu, 0, IDM_ABOUT, offset szMenuAbout
```

在窗口过程中处理系统菜单消息的分支结构为：

```
.elseif   eax ==    WM_SYSCOMMAND
mov       eax, wParam
.if       ax == 自定义 ID1
...
.elseif   ax == 自定义 ID2
...
.else
        invoke   DefWindowProc, hWnd, uMsg, wParam, lParam
        ret
.endif
```

与处理 WM_COMMAND 消息不同的是，在 WM_SYSCOMMAND 消息中处理了自定义的菜单命令 ID 后，必须把其他命令 ID 交给 DefWindowProc 处理，并直接把返回值返回给 Windows，不然的话会发现窗口不能移动，不能关闭，不能最小化……因为它相当于屏蔽了所有 SC_RESTORE, SC_MOVE, SC_SIZE, SC_MAXIMIZE, SC_MINIMIZE 和 SC_CLOSE 等消息的处理。

6. 右键弹出菜单

例子程序的一个功能是当用户在窗口客户区按下鼠标右键的时候弹出一个菜单，这个功能是用 TrackPopupMenu 函数实现的。TrackPopupMenu 函数的用法：

```
invoke    TrackPopupMenu, hMenu, uFlags, x, y, nReserved, hWnd, lpRect
```

这个函数本身很简单，执行后在参数指定的 x, y 位置弹出一个属于 `hWnd` 窗口（也就是说 `WM_COMMAND` 消息发到这个窗口）的菜单，菜单句柄是 `hMenu`。由于函数中的坐标是以整个屏幕左上角为基准的，所以弹出菜单的位置不一定在窗口的客户区内，它可以是屏幕的任何一个地方。

`uFlags` 参数指定一些和位置相关的选项，它可以是 `PM_CENTERALIGN`, `TPM_LEFTALIGN` 或 `TPM_RIGHTALIGN` 三者之一，表示 (x, y) 坐标是代表弹出菜单位置的中间、左上角还是右上角，一般的习惯是使用 `TPM_LEFTALIGN`，这样菜单会在鼠标点击处的右边弹出。`uFlags` 中同时还可以指定用鼠标左键还是右键选定菜单项，定义值可以是 `TPM_LEFTBUTTON` 或 `TPM_RIGHTBUTTON`，如果选择 `TPM_RIGHTBUTTON` 的话，对在菜单项上面按鼠标左键是没有反应的。

`lpRect` 指向一个 `RECT` 结构，用来指定一个区域，当菜单弹出后，在这个区域外单击鼠标，菜单才会消失，如果这个参数指定为 `NULL` 的话，在菜单之外单击鼠标，菜单就会消失。

在使用 `TrackPopupMenu` 之前，有几个准备工作是要做的：为了在客户区中按下鼠标右键弹出菜单，我们当然要处理鼠标右键消息，也就是说在 `WM_RBUTTONDOWN` 消息中调用 `TrackPopupMenu` 函数，一般的习惯是在鼠标按下的地方弹出菜单，所以还要首先获取鼠标光标的位置，然后在此位置弹出菜单。

要获取鼠标位置，可以用 `GetCursorPos` 函数：

invoke `GetCursorPos, lpPoint`

参数 `lpPoint` 指向一个 `POINT` 数据结构，这个结构只有两个字段：

```
POINT    STRUCT
    x     DWORD ?
    y     DWORD ?
POINT    ENDS
```

该结构用来表示一个点的 (x, y) 坐标，`GetCursorPos` 将当前的鼠标位置返回到这个结构中，程序中的相关代码是：

```
local    @stPos:POINT                                ;首先定义一个 POINT 结构
...
invoke   GetCursorPos, addr @stPos                ;获取鼠标位置
invoke   TrackPopupMenu, hSubMenu, \
         TPM_LEFTALIGN, @stPos.x, @stPos.y, NULL, hWnd, NULL
```

用 `GetCursorPos` 获取的鼠标位置是一个 `POINT` 结构，但由于 `TrackPopupMenu` 输入坐标的方法是用 x, y 两个参数，而不是一个 `POINT` 结构，所以要用结构中的两个字段 `@stPos.x` 和 `@stPos.y` 分别输入。

使用 `TrackPopupMenu` 时要注意的，弹出的菜单句柄必须是 `popup` 类型的，而在资源文件中定义并且可以用 `LoadMenu` 函数装入的菜单并不是 `popup` 类型的，`popup` 菜单（如例子中的“文件”与“查看”等）只能在第二层中才能定义，在程序中用 `GetSubMenu` 得

到的第二层子菜单的句柄才是 popup 类型的。GetSubMenu 函数的用法是：

```
invoke    GetSubMenu, hMenu, nPos
.if      eax
    mov    hSubMenu, eax
.endif
```

nPos 参数指定要获取的菜单项的位置索引，GetSubMenu 的返回值是获取的子菜单句柄。

例如用 invoke GetSubMenu, hMenu, 1 取得第二个子菜单（“文件”子菜单为 0，“查看”子菜单为 1，……）的句柄，然后在 TrackPopupMenu 中使用，这个菜单句柄就是主菜单中的“查看”菜单，所以按鼠标右键弹出的菜单和下拉菜单中的“查看”菜单是一模一样的。

7. 菜单状态的检测和设置

在程序中经常要对菜单项的状态进行设置，比如，剪贴板中没有数据时，“粘贴”菜单项应该灰化，窗口中没有被选中的字符时，“拷贝”菜单项也应该灰化，这样可以给使用者一个善意的提醒。同样，对菜单的状态也常常需要检测，如查看菜单项的状态是否处于灰化状态或选中状态以便进行下一步操作等。

对菜单项状态的检测可以用 GetMenuState 函数来完成，用法是：

```
invoke    GetMenuState, hMenu, uId, uFlags
```

参数 hMenu 是菜单的句柄，uId 用来定位要检测的菜单项，当 uFlags 是 MF_BYCOMMAND 的时候，uId 用菜单项的命令 ID 指定，当 uFlags 是 MF_BYPOSITION 的时候，uId 的值是位置索引，函数执行后的返回值为 -1 时表示失败，否则会是 MF_CHECKED, MF_DISABLED, MF_GRAYED, MF_HILITE, MF_MENUBARBREAK, MF_MENUBREAK 和 MF_SEPARATOR 的组合值，它们分别表示菜单项的状态是选中、禁用、灰化、高亮显示，以及 3 种分隔线，读者可以用 test 指令测试相应的数据位来分辨菜单项处于哪种状态，一般的测试代码如下：

```
invoke    GetMenuState, hMenu, IDM_XXX, MF_BYCOMMAND
.if      eax & MF_CHECKED
    ;表示 IDM_XXX 菜单项现在是选中状态
.endif
```

同样，读者也可以用 eax & MF_DISABLED 和 eax & MF_GRAYED 等条件测试其他状态。

设置菜单项的状态可以用下列 3 个函数来实现不同的功能：

```
invoke    EnableMenuItem, hMenu, uIDEnableItem, uEnable
invoke    CheckMenuItem, hMenu, uIDCheckItem, uCheck
invoke    CheckMenuRadioItem, hMenu, idFirst, idLast, idCheck, uFlags
```

EnableMenuItem 函数将菜单项在禁用、可用和灰化状态之间切换，uEnable 可以取值为 MF_DISABLED, MF_ENABLED 和 MF_GRAYED，它们分别代表这 3 种状态。

CheckMenuItem 函数将菜单项在非互斥的选定状态和非选定状态之间切换（即前面是否有对钩），uCheck 的取值可以是 MF_CHECKED 或 MF_UNCHECKED，代表选定或非选定状态。

CheckMenuRadioItem 将菜单项在互斥的选定状态和非选定状态之间切换（即前面是否

有圆点标志)，由于互斥的菜单项在一个范围内只有一个是可选定的，当选定另一个的时候，原来的选定应该撤销，`idFirst` 和 `idLast` 就指定了这个互斥范围。函数在选定 `idCheck` 指定的菜单项的同时将自动清除 `idFirst` 和 `idLast` 范围内的其他选定。所以 `uFlags` 中无须指定状态，只需指定 `MF_BYCOMMAND` 或 `MF_BYPOSITION` 定位方法。

在这些函数的参数中，`uIDEnableItem`，`uIDCheckItem`，`idFirst`，`idLast` 和 `idCheck` 用来定位菜单项，同样，参数的取值可以是菜单项的命令 ID 或位置索引，可以在状态参数（`uEnable`，`uCheck`，`uFlags`）中组合定义 `MF_BYCOMMAND` 或 `MF_BYPOSITION` 来决定使用哪种方法。

在例子程序中，当选中 `IDM_TOOLBAR` 和 `IDM_STATUSBAR` 之间的菜单项的时候，程序先用 `invoke GetMenuState, hMenu, ebx, MF_BYCOMMAND` 获取当前的状态，检查是否选定，并将选定状态反转后用 `CheckMenuItem` 重新设置：

```
.elseif    eax >=    IDM_TOOLBAR && eax <= IDM_STATUSBAR
            mov      ebx, eax
            invoke   GetMenuState, hMenu, ebx, MF_BYCOMMAND
            .if      eax ==    MF_CHECKED
                mov   eax, MF_UNCHECKED
            .else
                mov   eax, MF_CHECKED
            .endif
            invoke   CheckMenuItem, hMenu, ebx, eax
```

当选中 `IDM_BIG` 和 `IDM_DETAIL` 之间的菜单项的时候，程序用 `CheckMenuRadioItem` 将原先 `IDM_BIG` 和 `IDM_DETAIL` 范围内的互斥选定撤销并将当前选定的菜单项加圆点标记。

```
.elseif    eax >=    IDM_BIG && eax <= IDM_DETAIL
            invoke   CheckMenuRadioItem, hMenu, IDM_BIG, IDM_DETAIL, \
            eax, MF_BYCOMMAND
```

最后，修改菜单状态的时机是什么时候呢？在程序中似乎不应该随时去检测状态并设置，这显然是很浪费资源的。Windows 考虑到了这一点：在菜单将要激活的时候，也就是用户在菜单上按动鼠标的时候，Windows 在菜单弹出之前会向窗口过程发送 `WM_INITMENU` 消息，我们可以从容不迫地在这里进行各种检测，并设置对应的菜单项。



读者可以注意到，在状态参数中指定 `MF_BYCOMMAND` 或 `MF_BYPOSITION` 将决定位置参数用命令 ID 还是位置索引表示，这个规则在所有的菜单函数中都是适用的，`MF_BYCOMMAND` 是默认值（它的定义值是 0），如果两者都不定义的话，位置参数代表的就是命令 ID。

8. 其他菜单函数

除了前面介绍的一些函数之外, 还有一些不太常用的菜单函数, 在这里做简单的介绍。

菜单不一定非要在资源文件中定义, 在程序中也可以用代码来建立菜单, 不过比较麻烦一点的方法是先使用 `CreateMenu` 建立一个菜单, `CreateMenu` 函数没有参数, 调用后返回一个没有任何菜单项的菜单句柄, 接下来就可以用 `AppendMenu` 在上面一条条地添加菜单项了。

同样, `CreatePopupMenu` 也可以建立一个没有任何菜单项的菜单句柄, 但它建立的是 `popup` 类型的菜单句柄, 可以在 `TrackPopupMenu` 中直接使用。

如果要获取一个窗口当前使用的菜单句柄, 那么可以使用 `GetMenu` 函数:

invoke	<code>GetMenu, hWnd</code>
mov	<code>hMenu, eax</code>

一个菜单的总项数可以用 `GetMenuItemCount` 函数获取:

invoke	<code>GetMenuItemCount, hMenu</code>
--------	--------------------------------------

不过 `GetMenuItemCount` 函数的返回值不包括子菜单展开以后的项数, 而是指最上层菜单的项数。比如, 在例子程序中对 `hMenu` 统计的结果是 3, 因为 `Menu.rc` 中定义的最上层的菜单项是“文件”、“查看”和“帮助”, 总共 3 个, 如果要统计全部展开后的项数, 那么只好用 `GetSubMenu` 一层层地统计下去了。

对菜单中各个菜单项当前的文字和命令 ID 也是可以查询的, 方法是用 `GetMenuString` 和 `GetMenuItemID`, 读者可以自行参考命令手册。

建立窗口时指定了菜单句柄后并不是不能改变的, 我们常常见到一些编辑软件, 没有打开文件之前菜单只有寥寥几项, 一打开文件以后功能菜单就全部出来了, 实际上这是用 `SetMenu` 函数完成的:

invoke	<code>SetMenu, hWnd, hMenu</code>
--------	-----------------------------------

可以在资源文件中预定义几个不同的菜单, 在使用的时候根据不同情况随时用 `SetMenu` 设置不同的菜单句柄。

使用菜单后, 要涉及清除的问题, 与窗口相连的菜单句柄在窗口摧毁的时候会由 Windows 自动释放, 不需要手工释放, 但没有与窗口相连的菜单就要由程序自己来释放了, 方法是使用 `DestroyMenu` 函数, 比如没有与窗口相连而仅用 `TrackPopupMenu` 弹出的菜单句柄:

invoke	<code>DestroyMenu, hMenu</code>
--------	---------------------------------

5.2 图标和光标

图标和光标是图形资源, 图标通常用做应用程序的“形象代表”出现在文件浏览器、运行窗口左上角或程序的快捷方式等所有代表文件的地方, 为自己写的应用程序选一个合适的图标会使程序变得引人注目; 而光标就是鼠标移动时屏幕上那个指示位置的箭头, 应

用程序可以定义自己的光标，这样，光标移到程序的客户区中就会变成需要的形状。

5.2.1 图标和光标的资源定义

与菜单、加速键等资源不同，在资源脚本文件中定义图标和光标时并不是一个一个像素地定义，而是指定图标和光标的文件名，由资源编译器将像素数据读入再转换成二进制格式。由此可见，在资源定义之前要用其他工具先创建图标和光标文件。图标和静态光标文件的扩展名分别是 ico 和 cur，还有一种扩展名为 ani 的动态光标文件。

光标和图标在资源文件中的定义语句是：

图标 ID	ICON [DISCARDABLE]	图标文件名	;定义图标
光标 ID	CURSOR [DISCARDABLE]	光标文件名	;定义光标

DISCARDABLE 关键字是内存选项，表示在不用的时候可以从内存暂时卸掉，当文件名包含空格时，两边要用双引号引起来，图标 ID 和光标 ID 同样也可以用 16 位的整数或字符串表示，这里是几个定义的例子：

MyIcon icon	"1.ico"	;把 1.ico 定义为 ID 为 "MyIcon" 的图标资源
1000 icon	discardable 2.ico	;把 2.ico 定义为 ID 为 1000 的图标资源
1001 icon	"big icon.ico"	;把 big icon.ico 定义为 ID 为 1001 的图标资源
1002 cursor	"big arrow.ani"	;把 big arrow.ani 定义为 ID 为 1002 的光标资源
GoodCursor cursor	arrow.cur	;把 arrow.cur 定义为 ID 为 "GoodCursor" 的光标资源



资源文件中定义的图标可以不止一个，但由于 Windows 在“我的电脑”中列出文件的时候总是使用资源中的第一个图标当做文件的图标，所以在资源脚本文件中要把想用做程序图标的图标定义语句排在最前面。

5.2.2 使用图标和光标

在这里，用一个例子来说明图标和光标的用法，程序是建立在 FirstWindow.asm 和 Menu.asm 程序的基础上的，为了节省篇幅，在这里就不列出全部源程序了，完整的源程序可以在所附光盘的 Chapter05\Icon 目录中找到。程序中创建了一个菜单，运行后可以在“图标和光标”菜单中选择不同的图标和光标，选择不同的图标以后，窗口标题栏左边的图标和桌面任务栏上的窗口图标都会变化；选择不同的光标后，当鼠标移动到窗口客户区中的时候，光标会变成程序指定的光标。具体的效果如图 5.3 所示，大图标对应“笑脸”，小图标对应“箭头”，而光标 A 和 B 分别是“小恐龙”光标和“手型”光标，其中“小恐龙”光标是 ani 类型的动态光标，在屏幕上显示为一个走动中的恐龙模样。



图 5.3 不同的图标和光标

[illegible]

Icon.asm 的大部分是窗口模板程序的内容,与 FirstWindow.asm 是相同的,仅在窗口过程的 WM_CREATE 和 WM_COMMAND 增加了一些内容:

140

```

        mov     hCur1, eax
        invoke  LoadCursor, hInstance, CUR_2
        mov     hCur2, eax
        invoke  SendMessage, hWnd, WM_COMMAND, IDM_BIG, NULL
        invoke  SendMessage, hWnd, WM_COMMAND, IDM_CUR1, NULL
    .elseif    eax == WM_COMMAND
        mov     eax, wParam
        movzx   eax, ax
        .if     eax == IDM_EXIT
            call _Quit
        .elseif    eax == IDM_BIG
            invoke  SendMessage, hWnd, WM_SETICON, ICON_BIG, hIcoBig
            invoke  CheckMenuItem, hMenu, \
                IDM_BIG, IDM_SMALL, IDM_BIG, MF_BYCOMMAND
        .elseif    eax == IDM_SMALL
            invoke  SendMessage, hWnd, \
                WM_SETICON, ICON_BIG, hIcoSmall
            invoke  CheckMenuItem, hMenu, \
                IDM_BIG, IDM_SMALL, IDM_SMALL, MF_BYCOMMAND
        .elseif    eax == IDM_CUR1
            invoke  SetClassLong, hWnd, GCL_HCURSOR, hCur1
            invoke  CheckMenuItem, hMenu, \
                IDM_CUR1, IDM_CUR2, IDM_CUR1, MF_BYCOMMAND
        .elseif    eax == IDM_CUR2
            invoke  SetClassLong, hWnd, GCL_HCURSOR, hCur2
            invoke  CheckMenuItem, hMenu, \
                IDM_CUR1, IDM_CUR2, IDM_CUR2, MF_BYCOMMAND
    .endif

```

1. 装入图标和光标

在 WM_CREATE 消息中，程序从资源节区中装入所有的图标和光标资源，装入图标是用 LoadIcon 函数来完成的：

```

        invoke  LoadIcon, hInstance, lpIconName
    .if     eax
        mov     hIcon, eax
    .endif

```

hInstance 参数指定实例句柄，表示图标资源定义在哪个可执行文件中，lpIconName 参数指定图标资源的名称，它就是资源文件中定义的图标 ID 值，如果调用成功的话，函数返回图标句柄。

除了可以装入资源文件中定义的图标资源之外，当参数 hInstance 为 NULL 的时候，用 LoadIcon 还可以用预定义的 lpIconName 参数装入 Windows 预定义的图标，参数说明如表 5.1 所示。

表 5.1 LoadIcon 可以装入的预定义图标

lpIconName 参数的预定义值	图标形状
IDI_APPLICATION	应用程序默认图标
IDI_ASTERISK	I 符号图标

续表

lpIconName 参数的预定义值	图标形状
IDI_EXCLAMATION	警告图标（黄色三角形+感叹号）
IDI_HAND	严重警告图标（一般是红色圆形+叉）
IDI_QUESTION	问号图标
IDI_WINLOGO	Windows 标徽图标

装入光标的函数有两个。装入在资源中定义的光标的函数是 LoadCursor，它的语法和 LoadIcon 几乎一样：

invoke	LoadCursor, hInstance, lpCursorName
.if	eax
	mov hCursor, eax
.endif	

LoadCursor 的用法也和 LoadIcon 相似，lpCursorName 是光标资源的 ID，LoadCursor 也可以用指定 hInstance 为 NULL 的办法装入表 5.2 所列的预定义光标，这时候 lpCursorName 参数的取值如表 5.2 所示。

表 5.2 LoadCursor 可以装入的预定义光标

预定义值	光标形状
IDC_APPSTARTING	标准的箭头形状加上小沙漏
IDC_ARROW	标准的箭头形状
IDC_CROSS	十字型光标
IDC_IBEAM	要求输入文字时的 I 型光标
IDC_NO	禁止光标（圆圈里面加一个斜杠）
IDC_SIZE	改变大小时的十字箭头
IDC_SIZENESW	东北和西南方向的双向箭头
IDC_SIZENS	向北和向南的双向箭头
IDC_SIZESE	西北和东南方向的双向箭头
IDC_SIZESW	向西和向东的双向箭头
IDC_UPARROW	垂直箭头光标
IDC_WAIT	沙漏光标



读者可以注意到，预定义的图标和光标都是 Windows 系统中常用的，预定义图标常用在消息框中，预定义光标就是 Windows 鼠标属性中的光标。使用预定义图标和光标的好处是它们的形状会随着系统设置值的不同自动改变，如改变“控制面板”→“鼠标”→“指针”中的设置后，装入的光标会自动改变。

另一个光标装入函数是 LoadCursorFromFile，这个函数从磁盘光标文件中装入光标：

invoke	LoadCursorFromFile, lpCursorFileName
.if	eax
	mov hCursor, eax

```
.endif
```

在 Windows 9x 中，静态光标文件*.cur 既可以定义在资源文件中，也可以使用 LoadCursorFromFile 函数装入，但是动态光标文件*.ani 只能通过文件方式装入。在 Windows 2000 及 XP 中，两种光标文件都可以通过资源装入。为了在不同的操作系统上都可以使用，例子文件使用 LoadCursorFromFile 函数来装入动态光标文件。

2. 使用图标和光标

现在来看如何使用图标。图标一般使用在对话框中或者程序窗口的标题栏中，要在标题栏中设置图标可以用向窗口发送 WM_SETICON 消息的办法实现：

```
invoke    SendMessage, hWnd, WM_SETICON, ICON_BIG, hIcon
```

消息的 wParam 参数可以是 ICON_BIG 或 ICON_SMALL，用来指定图标的分辨率为 32×32 还是 16×16。

要将窗口的光标设置为新的光标不能使用 WM_SETCURSOR，这个消息是通知窗口重新刷新光标而不是让它设定指定的光标。Windows 中有个 SetCursor 函数可以用来设置窗口光标，但这只能将新的光标维持很短一段时间，因为当 Windows 向窗口重新发送 WM_SETCURSOR 消息的时候，光标就会被设置为原来的样子，而不妙的是，Windows 常常会自动向窗口过程发送 WM_SETCURSOR 消息，所以 SetCursor 并不能用来永久地改变窗口的光标。

如果要改变窗口的光标，正确的办法是用 SetClassLong 函数改变窗口类的属性，这个函数的使用方法如下：

```
invoke    SetClassLong, hWnd, nIndex, dwNewLong
```

由于这个函数用来改变窗口类的属性，所以可以永久改变类中的光标设定，hWnd 用来指定一个用这个类建立的某个窗口句柄，nIndex 参数指定要改变窗口类的哪个属性，可以指定为 GCL_HBRBACKGROUND, GCL_HCURSOR, GCL_HICON, GCL_HMODULE, GCL_MENU, GCL_STYLE 或 GCL_WNDPROC 等，它们分别表示要改变的窗口类的背景色、光标、图标、hInstance、菜单、风格或窗口过程地址，读者可以用这个函数来改变一个窗口类的几乎所有属性，程序中通过这个函数将窗口的光标在不同的光标句柄之间切换：

```
invoke    SetClassLong, hWnd, GCL_HCURSOR, hCur1 或 hCur2
```

5.3 位图

5.3.1 位图简介

位图 (Bitmap) 是 Windows 操作系统存储图像的方式，图像中的每个像素对应存储器中的一个或多个数据位，如单色位图每个像素对应 1 位，16 色位图每个像素对应 4 位，256 色为 8 位，全彩色为 24 位等，所有的像素数据按照一行行的顺序排列在存储器中，每个像素对应的位数称为颜色深度。

使用位图的优越之处是操作的速度很快，计算机的屏幕显示是由硬件从视频缓冲区中的数据映射的，向视频缓冲区中拷贝数据就可以直接将图像显示在屏幕上，所以以位图的方式存储像素，显示图形的时间几乎就是向视频缓冲区拷贝数据的时间。

位图的不便之处一是尺寸问题：由于位图是不压缩的，它占用的空间很大，一个 $1\,024 \times 768$ 像素、24 位色的图像的大小为 $1\,024 \times 768 \times 3$ ，达 2.3 MB；二是位图的缩放问题，读者都知道矢量和位图之间的关系，矢量图形（网上流行的 Flash 动画就是矢量格式的）可以无限制缩放而不变形，因为它是根据矢量实时计算出像素数据的，而位图缩放后要对原来的像素数据进行插值计算，不可避免地会有失真。

在使用 Windows 的位图之前，必须搞清楚几个概念：位图、设备无关位图和位图文件。

单纯意义上的“位图”指的就是存放在内存中、可以马上使用的位图，它的颜色深度总是对应当前设备（如屏幕或打印机等）的颜色深度。不与具体的设备对应，位图数据是没有意义的，因为无法知道要把数据中的多少位解释成一个像素。

对于存放在磁盘上或别的地方的位图数据来说，它的颜色深度有可能和屏幕颜色深度不同，为了准确描述它的颜色信息，必须有像素数据的属性说明，以及色彩表，在使用这个位图的时候，可以根据这些信息将像素数据转换到需要的颜色深度。色彩表和位图数据合在一起就叫做设备无关位图（DIB），因为它转换后可以用于不同颜色深度的设备上。Windows 有函数专门用来处理 DIB。使用 DIB 惟一的问题是当将高颜色数的 DIB 转换到低颜色数的设备上时，由于色彩只能被转换成设备所能表示的最相近的颜色，所以可能会有很大的颜色失真。

DIB 可以存放在磁盘上的位图文件中，位图文件一般以 bmp 为扩展名，它的内容包括一个 bitmap 文件头和 DIB 数据，bitmap 文件头可以用来验证整个文件的有效性。所以简单地讲，DIB 是位图数据的超集，位图文件又是 DIB 的超集。

Windows 支持的图形文件格式只有 bmp、ico 和 cur 等几种，可以广泛用在 GDI 操作中的只有 bmp 文件，其他格式文件，如 jpg 与 tif 等都是不能直接应用的，要使用这些文件，必须在代码中将它们转换到位图格式以后才行，所以要编一个仅支持 bmp 的图片浏览器是很简单的，而要支持其他格式麻烦就大了，仅 jpg 格式的解码就是个很复杂的问题！

5.3.2 在资源中定义位图

Windows 对 bmp 文件的支持有两种方法，一种是打开 bmp 文件读入 DIB 部分的数据，然后用函数将 DIB 数据转换到位图数据；另一种方法就是在资源文件中用和 ico、cur 文件类似的方法定义位图资源，然后在程序中装入后使用。

在资源脚本文件中定义位图资源的语法是：

位图 ID BITMAP [DISCARDABLE] 位图文件名

在程序中可以用 LoadBitmap 函数装入位图资源：

```
invoke    LoadBitmap, hInstance, lpBitmapName
.if        eax
```

```
        mov     hBitmap, eax  
.endif
```

LoadBitmap 函数返回一个位图句柄，在程序退出的时候，位图句柄必须用 DeleteObject 函数释放。对位图资源的大部分操作涉及 GDI 的内容，这方面的内容在第 7 章中详细介绍。

5.4 对话框

5.4.1 对话框简介

顾名思义，对话框完成的是“对话”功能，应用程序一般建立一个主窗口用做工作界面，大部分的工作会在主窗口的客户区完成，但程序往往需要和用户交互，如输入参数和输入文本等，这些界面不必要全部放在主窗口中。习惯的做法是通过选择菜单项弹出一个窗口，然后在这个窗口中完成对话，这个窗口就是“对话框”，对话框中的按钮、文本框和图标等称为“子窗口控件”。

为了提示用户，习惯于在会引出对话框的菜单项后面加上省略号。如“文件”菜单中的“另存为...”表示会引出一个选择文件名的对话框，所以“另存为”3个字后面加了个省略号。对话框最典型的例子就是写字板“查找”菜单弹出的窗口，以及 IE 浏览器中选择“Internet 选项”菜单项弹出的设置窗口。

1. 对话框的类型

对话框分两类：modal 对话框和 modeless 对话框，翻译成中文就是“模态的”和“非模态的”（也有的地方翻译成“模式的”和“非模式的”，Visual FoxPro 中文版就是这样），它们之间的区别在于是否允许用户在不同窗口间进行切换：当显示非模态对话框时，用户可以随意在这个对话框和其他窗口之间切换；而显示一个模态对话框时，用户在关闭对话框之前不允许切换到同一程序的其他窗口中，但可以切换到其他程序的窗口中；如果显示的是操作系统所属的模态对话框（即“系统模态的”），则切换到其他任何程序的窗口都是不允许的。

Windows 在资源文件中定义对话框，然后在程序中利用这个模板创建对话框，模态对话框和非模态对话框的资源定义是相同的，只是创建时调用的函数不同而已。

2. 对话框的工作原理

很明显，对话框和普通窗口之间有很多相似之处，实际上对话框就是基于窗口的，对话框的窗口风格使用的就是普通窗口的风格定义，对话框也有一个类似于窗口过程的对话框过程，但对话框和普通窗口在实现上又有很多不同之处，模态对话框和非模态对话框的实现也是不同的，图 5.4 对比了它们之间的不同之处。普通的窗口在建立之前需要用 RegisterClass 注册一个窗口类，然后用 CreateWindow 建立窗口，建立窗口所需的参数如窗口风格、大小位置和窗口过程地址等由窗口类，以及 CreateWindow 中的参数共同提供。

建立对话框的时候并不使用 `CreateWindow` 函数，取而代之，建立模态对话框的函数是 `DialogBoxParam`，建立非模态对话框的函数是 `CreateDialogParam`，调用这两个函数创建对话框窗口之前不需要注册对话框的窗口类。

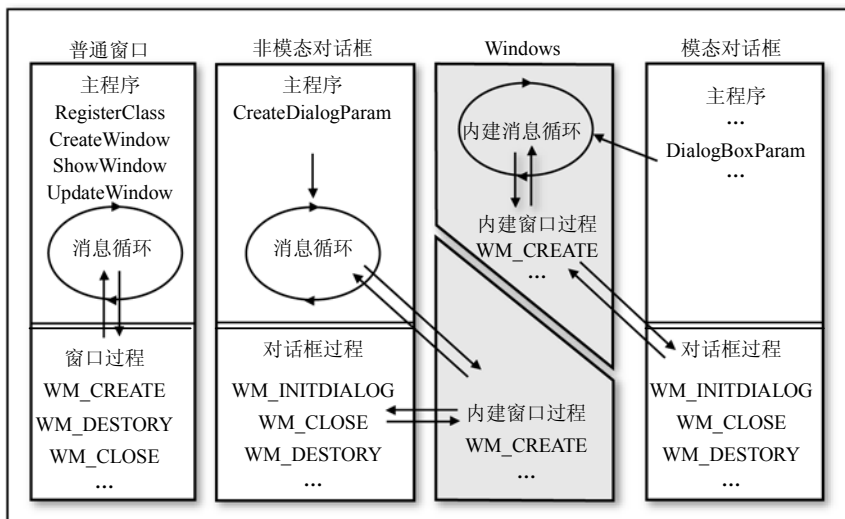


图 5.4 对话框和普通窗口工作方式的区别

Windows 在这两个函数的内部调用 `CreateWindowEx` 来建立对话框，使用的风格、大小和位置等参数取自资源中定义的对话框模板，使用的窗口类则是 Windows 内部定义的类。如果读者用一些工具去查看，会发现类名是“#32770”之类的字符串，在这个名字奇特的窗口类中，窗口过程被定义到了 Windows 内部的“对话框管理器”代码中，Windows 在这里处理对话框的大部分消息，如维护客户区的刷新，键盘接口（按 Tab 键在不同子窗口之间切换、按回车键调用默认按钮等），对话框管理器在初始化对话框时会根据对话框模板中定义的子窗口控件建立对话框中所有的子窗口。

用户程序中的对话框过程是由对话框管理器调用的，在处理消息前，对话框管理器会先调用用户指定的对话框过程，再根据对话框过程的返回值决定是否处理它们。

Windows 对模态对话框和非模态对话框的处理有些不同。在创建并显示模态对话框后，Windows 会为它在内部建立一个消息循环，在这个消息循环中把消息发送给对话框管理器，对话框管理器在处理消息的过程中会调用用户定义的对话框过程，当对话框关闭的时候，Windows 退出内建的消息循环，并从 `DialogBoxParam` 函数返回。而对于非模态对话框，`CreateDialogParam` 函数在创建对话框后直接返回，对话框窗口的消息是通过用户程序中的消息循环派送的。

由于模态对话框的特征，使得用它来做小程序的主窗口非常方便，因为用一句 `DialogBoxParam` 函数就可以搞定了，既不用注册窗口类，也不用写消息循环，这对看到创建窗口的几十句代码就烦的读者来说可真是个福音，笔者也很喜欢用模态对话框做程序的主窗口。这种方法的缺点就是无法使用依赖消息循环来完成的功能，很明显，加速键就不

在接下来的内容中,以一个最简单的例子来讲解如何实现模态对话框,所有的源程序可以在所附光盘的Chapter05\Dialog 目录中找到,包括资源脚本文件Dialog.rc,汇编源文件Dialog.asm以及makefile文件,Dialog.exe运行的结果如图5.5所示。



5.4.2 对话框的资源定义

在资源脚本中定义对话框的语法是:

对话框中的子窗口控件语句定义在 BEGIN/END（当然也可以用花括号）之中，在这之前，可以定义对话框的一些可选属性，每种属性单独用一行定义，常用的可选属性如表 5.3 所示。

属 性	定义语法	说 明
标题文字	CAPTION “文字”	定义显示在窗口标题栏上的文字
窗口类	CLASS “类名”	定义对话框窗口使用的窗口类，如果不定义，则使用 Windows 内建的类
窗口风格	STYLE 风格组合	定义对话框的窗口风格，同 CreateWindowEx 中的 dwStyle 参数
扩展风格	EXSTYLE 风格组合	定义对话框的扩展窗口风格，同 CreateWindowEx 中的 dwExStyle 参数
字体	FONT 大小，“字体名”	定义对话框包括子窗口控件使用的字体
菜单	MENU 菜单 ID	对话框中使用的菜单，菜单 ID 在同一个资源脚本文件中定义

```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#include                <resource.h>
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#define    ICO_MAIN        0x1000    //图标
#define    DLG_MAIN        1
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN            ICON "Main.ico"
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
DLG_MAIN DIALOG 50, 50, 113, 64
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "对话框模板"
FONT 9, "宋体"
{
```

```

ICON ICO_MAIN, -1, 10, 11, 18, 21
CTEXT "简单的对话框例子\n用 Win32ASM 编写", -1, 36, 14, 70, 19
DEFPUSHBUTTON "退出(&X)", IDOK, 58, 46, 50, 14
CONTROL "", -1, "Static", SS_ETCHEDHORZ | WS_CHILD | WS_VISIBLE, 6, 39, 103, 1
}

```

脚本文件中除了定义图标以外，另外还定义了一个 ID 为 1 的对话框，对话框中有 4 个子窗口控件，分别是图标、文本、按钮和一条横线，按钮的 ID 为 IDOK，其他的子窗口控件由于是静态控件，不会向对话框过程发送命令，所以 ID 就设置为 -1，这些控件的具体用法将在后面的内容中详细介绍。

定义中还指定了一些可选属性，STYLE 语句定义了对话框窗口的风格，CAPTION 语句把标题定义为“对话框模板”，FONT 语句指定了对话框使用的字体是大小为 9 的宋体。

对话框的位置为 (50, 50)，大小为宽 113 单位、高 64 单位，读者可能已经注意到：这个对话框的大小好像比宽 113 像素、高 64 像素的窗口要大。事实上的确如此，这也正是大小是“单位”而不是“像素”的原因。对话框的位置、大小以及所有子窗口控件的度量单位是根据系统字体的大小来决定的，横向（x 坐标和宽度）每单位为系统字符平均宽度的 1/4，纵向（y 坐标和高度）每单位为字符平均高度的 1/8，由于系统字体的字符高度大致为宽度的两倍，所以虽然这种计算方法有些费解，但横向和纵向的数值在视觉上还是成比例的，但和以“像素”为单位在数值上肯定是不同的。如果读者一定要知道这个值换算成像素后是多少，那么可以用 GetDialogBaseUnits 函数来获取系统字体的高度和宽度再进行计算。



当一些英文版的软件在中文 Windows 上运行的时候，对话框中有些文本往往被砍掉了尾巴，原因就是这些程序是在英文 Windows 上调试的，文本框的尺寸是以英文 Windows 系统字符的大小来度量的，到了其他语言的 Windows 上后，系统字符的大小可能改变，对话框的大小也随着改变，结果就是原来刚好的宽度可能会变得不够，这也算是对话框尺寸度量方法的缺点吧！

使用文本编辑器直接书写对话框脚本定义不是很直观，所以在创建对话框资源时最好使用可视化的资源编辑器，如 VC++ 或 ResourceWorkshop 等。

在子窗口控件的 ID 定义中有两个特殊的 ID 值——IDOK 和 IDCANCEL，在 Resource.h 中它们的值定义为 1 和 2，IDOK 是默认的“确定”ID，IDCANCEL 是默认的“取消”ID。如果一个按钮的 ID 是 IDOK，当焦点没有停留在其他按钮上的时候，在任何地方按下回车键就相当于按下了这个按钮，而按下 Esc 键的时候，就相当于按下了 ID 为 IDCANCEL 的按钮。

2. Tab 停留位和组

对话框中可以定义多个子窗口控件，有的子窗口控件可以拥有输入焦点（如按钮、文本框与组合框等），有些则不能（如图标与文本等），当对话框中有多个允许拥有输入焦点的子窗口控件时（有 WS_TABSTOP 风格），用户可以用 Tab 键将输入焦点切换到下一个有 WS_TABSTOP 风格的子窗口控件上，也可以用 Shift+Tab 键切换到上一个，Tab 键切换的

Tab 停留位并不是系统根据子窗口控件的坐标位置自动排列的，而是按照子窗口控件在资源脚本文件中的定义顺序来排列的，所以读者在定义的时候最好根据子窗口控件的位置适当排列语句的先后，以免按动 Tab 键切换的时候焦点上下左右无规则地跳来跳去。如果使用可视化的资源编辑器，那么菜单中一般会有“Tab 停留位”菜单项，在编辑完成后也要进到这个菜单项中设置一下，资源编辑器会根据设置调整 rc 文件中定义语句的先后顺序。

器印

使用对话框的代码分为创建部分和对话框过程两个部分。先看 Dialog.asm 的源代码,再分析具体的使用过程,源代码如下:

读者可以发现，相对于普通窗口的使用，对话框的使用显得特别简单，最明显的区别是主程序中的一大堆代码不见了，换成了一个 `DialogBoxParam` 语句。

创建模态对话框的函数是 `DialogBoxParam`，它的使用方法是：

函数的各参数说明如下:

- 要结束模态对话框，必须在对话框过程的 WM_CLOSE 消息中使用 EndDialog 函数：

150

不能使用通常的 DestroyWindow 函数，参数中的 hDlg 就是对话框窗口的句柄，dwResult 参数是退出时的返回值，这个值最后由 DialogBoxParam 函数返回到主程序中。

2. 创建非模态对话框

创建非模态对话框的函数是 CreateDialogParam，它的参数定义和 DialogBoxParam 一模一样：

```
invoke    CreateDialogParam, hInstance, lpTemplateName, hWndParent, \
          lpDialogFunc, dwInitParam
mov       hDlg, eax
```

CreateDialogParam 和 DialogBoxParam 在使用中有几个不同点：

- CreateDialogParam 在创建对话框后，会根据对话框模板的风格是否定义了 WS_VISIBLE 来决定是否显示对话框窗口。如果定义了则显示，没有的话，则程序需要在以后自行调用 ShowWindow 来显示它；而 DialogBoxParam 函数不管是否定义了 WS_VISIBLE 风格都会显示对话框。
- CreateDialogParam 在建立对话框窗口后直接返回，返回值是对话框窗口的句柄；而 DialogBoxParam 要在对话框关闭后才返回，返回值是 EndDialog 中的 dwResult 参数。
- 由于在 CreateDialogParam 返回后，应用程序在自己的消息循环中获取对话框消息，所以如果要用非模态对话框做程序的主窗口，消息循环的代码还是要写的；而 DialogBoxParam 是使用 Windows 为它内建的消息循环。
- 关闭非模态对话框仍然使用 DestroyWindow 函数，注意在这里不要用 EndDialog 函数。

3. 对话框过程

Windows 在“对话框管理器”——也就是为对话框内建的窗口过程中处理对话框消息，在处理前会首先调用用户定义的对话框过程，程序可以在这里选择是否自行处理某些消息。读者在理解时可以把“对话框管理器”看成是对话框的 DefWindowProc，凡是自己不想处理的消息都由它来处理。

与窗口过程一样，对话框过程是一个“回调”子程序，它由程序定义，Windows 来调用，模态对话框和非模态对话框的对话框过程是一样的。

对话框过程和窗口过程的输入参数是一样的，也是：

```
DialogProc  proc  hWndDlg, uMsg, wParam, lParam
```

在程序里面一般编写对话框过程的分支结构如下：

```
_ProcDlgMain  proc  uses ebx edi esi hWnd, wMsg, wParam, lParam

                mov     eax, wMsg
                .if     eax == WM_CLOSE
                    ;模态对话框用 EndDialog 关闭
```

```

;非模态对话框用 DestroyWindow 关闭
.elseif eax == WM_INITDIALOG
;初始化代码
.elseif eax == WM_COMMAND
;子窗口控件发送的消息
;wParam 的低 16 位为子窗口控件 ID
.elseif eax == WM_XXXX
;处理其他需要处理的消息
.else
    mov     eax, FALSE
    ret
.endif
mov     eax, TRUE
ret

```

```
_ProcDlgMain    endp
```

注意对话框过程和普通的窗口过程在使用上有以下区别：

- 窗口过程对应于不同的消息有各种不同含义的返回值，而对话框过程返回 BOOL 类型的值，返回 TRUE 表示已经处理了某条消息，返回 FALSE 表示没有处理。“对话框管理器”代码会根据返回值决定是否继续处理某一条消息（惟一的例外是 WM_INITDIALOG 消息）。
- 对于不处理的消息，不需要调用 DefWindowProc 来处理，这事情由“对话框管理器”来做。

“对话框管理器”不会把 WM_CREATE 消息转发给对话框过程，取而代之，它会以 WM_INITDIALOG 消息来调用对话框过程，程序可以在这里进行一些初始化的操作，WM_INITDIALOG 消息的返回值有点特殊，如果程序想自行设置输入焦点，那么可以用 SetFocus 函数把输入焦点设置到需要的子窗口控件上，然后返回 FALSE；如果返回 TRUE 的话，那么 Windows 会自动将输入焦点设置到第一个有 WS_TABSTOP 的子窗口控件上。

对话框过程在 WM_COMMAND 消息中处理子窗口控件发送的命令，当用户在对话框中按下了按钮，输入文字或选择复选框等操作时，子窗口控件会向对话框过程发送 WM_COMMAND 消息，wParam 是子窗口控件的 ID，如例子程序中处理“退出”按钮的消息，在里面用 EndDialog 函数关闭对话框。

对话框窗口的标题栏上默认没有定义图标，如果要像普通窗口一样显示一个图标，那么可以像例子程序中那样，在 WM_INITDIALOG 中用 WM_SETICON 消息来设置。

5.4.4 在对话框中使用子窗口控件

子窗口控件是一些 Windows 预定义类，它们实际上就是一个以对话框为父窗口的子窗口。对于程序员来说，在对话框中使用它们的时候并不需要手工去逐一创建，只需要在对话框中定义就可以了，“对话框管理器”会在初始化对话框的时候，根据定义语句自动建立所有的子窗口。

1. 子窗口控件的定义

子窗口控件定义的一般语法是：

CONTROL 文本, ID, 类, 风格, x, y, 宽度, 高度[, 扩展风格]

“文本”指控件的初始化值，“ID”是子窗口向对话框过程发送 WM_COMMAND 中用的 ID 值，“类”可以是按钮 (Button)、静态 (Static)、编辑 (Edit)、滚动条 (ScrollBar)、列表框 (ListBox) 和组合框 (ComboBox)，这些类都是 Windows 系统中已经预定义的，“对话框管理器”在初始化的时候把每一条控件定义语句转换成下面的 CreateWindow 命令：

```
invoke CreateWindow, 类名, 文本, 风格, \
      x, y, 宽度, 高度, \
      对话框窗口句柄, ID, hInstance, NULL
```

正因为如此，所有可以用 CreateWindow 建立的子窗口都可以在资源中定义，只要知道要使用的类和风格就可以了。所以除了上面这些基本的类之外，对话框中还可以使用一些通用控件，如“日期” (SysDateTimePick32)、“月历” (SysMonthCal32)、“热键” (msctls_hotkey32) 和“列表” (SysListView32) 等，括号内是它们的类名，只要把定义语句的“类”写成对应的名称就可以了。

基于同一个预定义类的控件根据风格属性的不同，外表可能完全不同，如单选钮、复选框和分组框使用的类都是 Button 类，文本、图标框、位图框和线条等都是 Static 类。使用 CONTROL 语句定义的时候可能不是很直观，所以 Rc.exe 资源编译器允许使用另一种语法来书写控件定义：

控件名称 [文本,] ID, x, y, 宽度, 高度[, 风格][, 扩展风格]

这里使用“控件名称”而不是“类”是因为这个名称只是 Rc.exe 使用的缩写，并不是真正的 Windows 类的名称，“控件名称”由 Rc.exe 解释成“类”名，同时为它使用了几种默认的风格，定义语句中风格属性实际上是附加在默认风格上的，表 5.4 列出了每种控件使用的类和默认属性，除了表中列出的默认属性外，每种控件还被默认定义了 WS_CHILD 和 WS_VISIBLE 属性。

表 5.4 资源脚本中使用的控件名称

控件名称	说 明	基于的类	默认窗口风格
PUSHBUTTON	按钮	Button	BS_PUSHBUTTON, WS_TABSTOP
DEFPUSHBUTTON	默认按钮	Button	BS_DEFPUSHBUTTON, WS_TABSTOP
CHECKBOX	复选框	Button	BS_CHECKBOX, WS_TABSTOP
AUTOCHECKBOX	自动复选框	Button	BS_AUTOCHECKBOX, WS_TABSTOP
STATE3	3 态复选框	Button	BS_3STATE, WS_TABSTOP
AUTO3STATE	自动 3 态复选框	Button	BS_AUTO3STATE, WS_TABSTOP
RADIOBUTTON	单选钮	Button	BS_RADIOBUTTON, WS_TABSTOP
AUTORADIOBUTTON	自动单选钮	Button	BS_AUTORADIOBUTTON, WS_TABSTOP
GROUPBOX	分组框	Button	BS_GROUPBOX

续表

控件名称	说 明	基于的类	默认窗口风格
SCROLLBAR	滚动条	ScrollBar	SBS_HORZ
CTEXT	居中文本	Static	SS_CENTER, WS_GROUP
LTEXT	左对齐文本	Static	SS_LEFT, WS_GROUP
RTEXT	右对齐文本	Static	SS_RIGHT, WS_GROUP
ICON	图标框	Static	SS_ICON
EDITTEXT	文本编辑	Edit	ES_LEFT, WS_BORDER, WS_TABSTOP
COMBOBOX	组合框	ComboBox	CBS_SIMPLE, WS_TABSTOP
LISTBOX	列表框	ListBox	LBS_NOTIFY, WS_BORDER

看下面的例子:

GROUPBOX "选项",	-1,	55,	5,	120,	100
PUSHBUTTON "退出",	IDCANCEL,	255,	115,	50,	14

这两条语句和下面的语句编译后产生的二进制资源文件是一模一样的：

CONTROL	"选项",	-1,	"Button",	BS_GROUPBOX	WS_TABSTOP,	55,	5,	120,	100
CONTROL	"退出",	IDCANCEL,	"Button",	BS_PUSHBUTTON	WS_TABSTOP,	255,	115,	50,	14

第一种语句的用法比第二种语句不但要直观许多，而且不必书写默认的窗口风格。

当用到的控件没有缩写语法时，那就必须用 CONTROL 定义了，下面的两句分别定义了一条横线和—个图片框，它们并没有缩写的用法：

CONTROL "", -1, "Static", SS_ETCHEDHORZ	WS_CHILD	WS_VISIBLE, 60, 65, 110, 1
CONTROL BMP_ID, -1, "Static", SS_BITMAP	WS_CHILD	WS_VISIBLE, 5, 5, 40, 95

下面以一个例子来演示各种子窗口控件的用法，读者可以在所附光盘的 Chapter05\Control 目录中找到全部的源代码，其中的 Control.rc 文件如下：

```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#include                                <resource.h>
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#define ICO_MAIN                0x1000    //图标
#define DLG_MAIN                 1
#define IDB_1                    1
#define IDB_2                     2
#define IDC_ONTOP               101
#define IDC_SHOWBMP            102
#define IDC_ALOW                103
#define IDC_MODALFRAME          104
#define IDC_THICKFRAME           105
#define IDC_TITLETEXT           106
#define IDC_CUSTOMTEXT         107
#define IDC_BITMAP              108
#define IDC_SCROLL             109
#define IDC_VALUE              110
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN                        ICON "Main.ico"
IDB_1                          BITMAP "Picture1.bmp"
```

编译后的 Control.exe 运行后的界面如图 5.6 所示。

对话框窗口控制示例

选项

☐ 隐在最前面

☒ 显示图片

☒ 允许更换图片

☐ 模态边框 (Modal Frame)

☒ 可变速边框 (Thick Frame)

标题栏文字

Hello, World!

自定义文字:

请在此选择显示在标题栏上面的文字, 或者选择“自定义”后自行输入

更换图片 (C)

退出 (X)

33

程序有这些功能：按下“更换图片”按钮⑧可以切换图片框⑨的图片；在组合框⑤中


```

        mov     eax, wParam
    .if     eax == WM_CLOSE
        invoke  EndDialog, hWnd, NULL
        invoke  DeleteObject, hBmp1
        invoke  DeleteObject, hBmp2
    .elseif eax == WM_INITDIALOG
;*****
; 设置标题栏图标
;*****
        invoke  LoadIcon, hInstance, ICO_MAIN
        invoke  SendMessage, hWnd, WM_SETICON, ICON_BIG, eax
;*****
; 初始化组合框
;*****
        invoke  SendDlgItemMessage, hWnd, \
            IDC_TITLETEXT, CB_ADDSTRING, 0, addr szText1
        invoke  SendDlgItemMessage, hWnd, \
            IDC_TITLETEXT, CB_ADDSTRING, 0, addr szText2
        invoke  SendDlgItemMessage, hWnd, \
            IDC_TITLETEXT, CB_ADDSTRING, 0, addr szText3
        invoke  SendDlgItemMessage, hWnd, \
            IDC_TITLETEXT, CB_SETCURSEL, 0, 0
        invoke  GetDlgItem, hWnd, IDC_CUSTOMTEXT
        invoke  EnableWindow, eax, FALSE
        invoke  LoadBitmap, hInstance, IDB_1
        mov     hBmp1, eax
        invoke  LoadBitmap, hInstance, IDB_2
        mov     hBmp2, eax
;*****
; 初始化单选按钮和复选框
;*****
        invoke  CheckDlgButton, hWnd, IDC_SHOWBMP, BST_CHECKED
        invoke  CheckDlgButton, hWnd, IDC_ALLOW, BST_CHECKED
        invoke  CheckDlgButton, hWnd, \
            IDC_THICKFRAME, BST_CHECKED
;*****
; 初始化滚动条
;*****
        invoke  SendDlgItemMessage, hWnd, \
            IDC_SCROLL, SBM_SETRANGE, 0, 100
;*****
    .elseif eax == WM_COMMAND
        mov     eax, wParam
;*****
; 由于印刷宽度有限，WM_COMMAND 消息处理代码的缩进格式有所影响，请读者注意
;*****
    .if     ax == IDCANCEL
        invoke  EndDialog, hWnd, NULL
        invoke  DeleteObject, hBmp1
        invoke  DeleteObject, hBmp2
;*****
; 更换图片
;*****

```

```

.elseif ax == IDOK
    mov     eax, hBmp1
    xchg    eax, hBmp2
    mov     hBmp1, eax
    invoke  SendDlgItemMessage, hWnd, IDC_BMP, STM_SETIMAGE, IMAGE_BITMAP, eax
;*****
; 设置是否“总在最前面”
;*****
.elseif ax == IDC_ONTOP
    invoke  IsDlgButtonChecked, hWnd, IDC_ONTOP
    .if     eax == BST_CHECKED
        invoke  SetWindowPos, hWnd, HWND_TOPMOST, 0, 0, 0, 0, \
            SWP_NOMOVE or SWP_NOSIZE
    .else
        invoke  SetWindowPos, hWnd, HWND_NOTOPMOST, 0, 0, 0, 0, \
            SWP_NOMOVE or SWP_NOSIZE
    .endif
;*****
; 演示隐藏和显示图片控件
;*****
.elseif ax == IDC_SHOWBMP
    invoke  GetDlgItem, hWnd, IDC_BMP
    mov     ebx, eax
    invoke  IsWindowVisible, ebx
    .if     eax
        invoke  ShowWindow, ebx, SW_HIDE
    .else
        invoke  ShowWindow, ebx, SW_SHOW
    .endif
;*****
; 演示允许和灰化“更换图片”按钮
;*****
.elseif ax == IDC_ALLOW
    invoke  IsDlgButtonChecked, hWnd, IDC_ALLOW
    .if     eax == BST_CHECKED
        mov     ebx, TRUE
    .else
        xor     ebx, ebx
    .endif
    invoke  GetDlgItem, hWnd, IDOK
    invoke  EnableWindow, eax, ebx
;*****
.elseif ax == IDC_MODALFRAME
    invoke  GetWindowLong, hWnd, GWL_STYLE
    and     eax, not WS_THICKFRAME
    invoke  SetWindowLong, hWnd, GWL_STYLE, eax
.elseif ax == IDC_THICKFRAME
    invoke  GetWindowLong, hWnd, GWL_STYLE
    or     eax, WS_THICKFRAME
    invoke  SetWindowLong, hWnd, GWL_STYLE, eax
;*****
; 演示处理下拉式组合框
;*****
.elseif ax == IDC_TITLETEXT

```

```

        shr     eax, 16
;*****
        .if     ax == CBN_SELENDOK
            invoke SendDlgItemMessage, hWnd, \
                IDC_TITLETEXT, CB_GETCURSEL, 0, 0
            .if     eax == 2
                invoke GetDlgItem, hWnd, IDC_CUSTOMTEXT
                invoke EnableWindow, eax, TRUE
            .else
                mov     ebx, eax
                invoke SendDlgItemMessage, hWnd, IDC_TITLETEXT, \
                    CB_GETLBTEXT, ebx, addr @szBuffer
                invoke SetWindowText, hWnd, addr @szBuffer
                invoke GetDlgItem, hWnd, IDC_CUSTOMTEXT
                invoke EnableWindow, eax, FALSE
            .endif
        .endif
;*****
; 在文本框中输入文字
;*****
        .elseif ax == IDC_CUSTOMTEXT
            invoke GetDlgItemText, hWnd, IDC_CUSTOMTEXT, \
                addr @szBuffer, sizeof @szBuffer
            invoke SetWindowText, hWnd, addr @szBuffer
        .endif
;*****
; 恢复代码的缩进格式, 请读者注意
;*****
; 处理滚动条消息
;*****
        .elseif eax == WM_HSCROLL
            mov     eax, wParam
            .if     ax == SB_LINELEFT
                dec     dwPos
            .elseif ax == SB_LINERIGHT
                inc     dwPos
            .elseif ax == SB_PAGELEFT
                sub     dwPos, 10
            .elseif ax == SB_PAGERIGHT
                add     dwPos, 10
            .elseif ax == SB_THUMBPOSITION || ax == SB_THUMBTRACK
                mov     eax, wParam
                shr     eax, 16
                mov     dwPos, eax
            .else
                mov     eax, TRUE
                ret
            .endif
            cmp     dwPos, 0
            jge     @F
            mov     dwPos, 0
            @@:
            cmp     dwPos, 100
            jle     @F

```

```

        mov     dwPos, 100
@@:
        invoke  SetDlgItemInt, hWnd, IDC_VALUE, dwPos, FALSE
        invoke  SendDlgItemMessage, hWnd, \
                IDC_SCROLL, SBM_SETPOS, dwPos, TRUE
;*****.else
        mov     eax, FALSE
        ret
    .endif
    mov     eax, TRUE
    ret

_ProcDlgMain    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
start:
        invoke  GetModuleHandle, NULL
        mov     hInstance, eax
        invoke  DialogBoxParam, hInstance, DLG_MAIN, \
                NULL, offset _ProcDlgMain, NULL
        invoke  ExitProcess, NULL
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        end     start

```

2. 子窗口控件的通用使用方法

由于子窗口控件实际上就是窗口，大部分窗口函数对它们都是适用的，如可以用 EnableWindow 在灰化和允许状态之间切换，可以用 ShowWindow 在显示和隐藏之间切换，可以用 GetWindowText 和 SetWindowText 来改变上面的文字，也可以用 MoveWindow 来改变大小和移动位置等。在 Control.asm 中用“显示图片”复选框切换图片框的隐藏和显示，用的就是 ShowWindow 函数，处理“允许更换图片”复选框时切换“更换图片”按钮的状态，用的是 EnableWindow 函数。

除了可以用对子窗口控件使用窗口的通用函数外，还可以使用针对它们的专用函数。下面介绍一些常用的函数。

在资源脚本文件中定义的是控件的 ID，当这些子窗口控件被创建以后同样会有一个窗口句柄，但既然它们不是由我们自己创建的，那么怎么知道它们的窗口句柄呢？有一个函数可以从 ID 中获取子窗口句柄：

invoke	GetDlgItem, hDlg, dwIDDlgItem
mov	hDlgItem, eax

函数的输入参数是对话框句柄和 ID 值，返回值是子窗口句柄；反过来，有两种方法可以从子窗口句柄获取 ID：

(1)	invoke	GetDlgCtrlID, hWndCtrl	;输入子窗口句柄，返回值是控件 ID
(2)	invoke	GetWindowLong, hWndCtrl, GWL_ID	

当需要向控件发送消息的时候，当然可以先用 GetDlgItem 获取子窗口句柄再用 SendMessage 函数，但有一个函数更为简便：

```
invoke    SendDlgItemMessage, hDlg, dwIDDlgItem, Msg, wParam, lParam
```

这个函数可以直接向控件发送消息，只需要在参数中指定对话框句柄和子窗口 ID。（注意：并没有 PostDlgItemMessage 这样的函数！）

如果要知道在一个控件上按下了 Tab 键或 Shift+Tab 键会跳到哪一个控件上去，也就是说下一个或上一个 Tab 停留位在哪里，可以使用 GetNextDlgTabItem 函数：

```
invoke    GetNextDlgTabItem, hDlg, hCtl, bPrevious
.if      eax
    mov    hWinNext, eax
.endif
```

其中的 bPrevious 参数指定了搜索的方向；与之相似，使用 GetNextDlgGroupItem 函数可以返回下一个分组的位置：

```
invoke    GetNextDlgGroupItem, hDlg, hCtl, bPrevious
.if      eax
    mov    hWinNext, eax
.endif
```

3. 使用单选钮和复选框

单选钮是互斥的选择钮，同一组的多个单选钮只能有一个被选中，单选钮的外形是一个圆形的标记加上文本，圆形中有黑点表示被选中。复选框不是互斥的，多个复选框的状态不会互相影响，复选框的外形是一个方框加上文本，方框中可以用有无打钩来表示是否被选中。

单选钮和复选框控件都是基于 Button 类的，只不过它们的窗口风格分别是 BS_RADIOBUTTON 和 BS_CHECKBOX。既然它们是特殊的“按钮”，所以和它们有关的函数都带有“Button”一词，查看一个单选钮或复选框是否被选中可以用下面的函数来检测：

```
invoke    IsDlgButtonChecked, hDlg, nIDButton
```

函数的返回值可能是 BST_CHECKED（选中状态），BST_INDETERMINATE（三态复选框的灰化状态）或 BST_UNCHECKED（未选中状态）。也可以用向子窗口控件发送 BM_GETCHECK 消息的方法来检测，返回值和上面的函数是一样的。

如果想设置单选钮或复选框的状态，可以使用下面的语句：

```
invoke    CheckDlgButton, hDlg, nIDButton, uCheck
```

参数 uCheck 用 BST_CHECKED, BST_INDETERMINATE 或 BST_UNCHECKED 来表示需要设置的状态，含义同上。向控件发送 BM_SETCHECK 消息也可以取得同样的效果，这时消息的 wParam 中放置需要设置的状态。

因为复选框是不互斥的，所以可以随意设置状态。而对于 BS_RADIOBUTTON 风格的单选钮来说，并不是把某个按钮设置为选中状态以后，同组的其他按钮就会自动变成非选中状态，所以用 CheckDlgButton 函数选中了一个单选钮以后，如果不是手动把同组的其他按钮全部改为非选中状态（逐个地调用 CheckDlgButton），就会看到同时有两个单选钮是

选中的。但由于把同组的所有单选按钮逐个地设置显得有点麻烦，所以针对单选按钮有一个专用函数：

invoke	CheckRadioButton, hDlg, \ nIDFirstButton, nIDLastButton, nIDCheckButton
--------	--

这个函数把 ID 在 nIDFirstButton 和 nIDLastButton 之间的单选按钮全部设置为非选中状态，只有 nIDCheckButton 是选中状态，当然在使用中要注意将这一批 ID 定义为连续的数值。

如果还嫌 CheckRadioButton 有点麻烦，还有一种最简单的办法——使用自动单选按钮，同组的 AUTORADIOBUTTON 会随着用户选中一个而自动清除其他单选按钮的状态，所以在程序中只需要在初始化的时候预设一次，其他时间就可以不必关心设置问题了，以后惟一用到的就是调用 IsDlgButtonChecked 检查状态了。

4. 使用静态控件

静态控件是基于 Static 类的子窗口控件，之所以叫“静态”控件，是因为它是“安静”的——它们不向对话框发送 WM_COMMAND 消息，所以静态控件的 ID 一般是没有用处的，定义时常常将它们定为-1，如果需要在程序中改变静态控件的属性，那么也可以为静态控件指定一个惟一的 ID 以便操作。

资源脚本文件中可以使用缩写的基于 Static 类的有 LTEXT, CTEXT, RTEXT（这是 3 种对齐方向不同的文本框）和 ICON（图标框），除了这些常用的类型之外，Static 类还可以用 CONTROL 语句通过指定不同的窗口风格派生出不同用途的控件来。

下面说明静态控件的一些用法。

对于文本框，文本长度超过边界的时候默认是自动换行的，但如果同时指定 SS_SIMPLE 风格的话，就不会自动换行。读者可以在程序中用 SetWindowText 或发送 WM_SETTEXT 消息来动态改变显示的文本，同样，也可以用 GetWindowText 或发送 WM_GETTEXT 消息来获取其中的文本。

静态控件可以用来构筑简单的线条和图形，如果指定 SS_BLACKFRAME, SS_GRAYFRAME 或 SS_WHITEFRAME 风格，那么静态控件显示为填充的矩形，填充颜色分别是黑色、灰色或白色；而指定 SS_BLACKRECT, SS_GRAYRECT 或 SS_WHITERECT 风格的话，则显示为非填充的矩形框，边线颜色是黑色、灰色或白色。

静态控件也可以用来做立体感的线条或边框，指定 SS_ETCHEDHORZ 风格显示为横线，指定 SS_ETCHEDVERT 风格显示为竖线，指定 SS_ETCHEDFRAME 风格则显示为立体的矩形框，视觉上的效果类似于没有文字的 GROUPBOX。

静态控件还有一个用途是做图形显示，当图形是图标的时候，用 ICON 语句就可以定义了，其默认的风格是 SS_ICON，如果想使用位图，那么可以指定 SS_BITMAP 风格，例子程序中的图片框就是这样定义的：

CONTROL	IDB_1, IDC_BMP, "Static", SS_BITMAP WS_CHILD WS_VISIBLE, 5, 5, 40, 95
---------	--

在这里,“文字”部分指定图资源的 ID,前面已经把 Picture1.bmp 的资源 ID 定义为 IDB_1, IDC_BMP 是图片框自己的 ID,如果不需要在程序中改变图片的话,那么这里可以定义为-1。

在程序中可以通过向控件发送 STM_SETIMAGE 消息来设置新的图片,消息的 wParam 指定图片的格式,取值可以是 IMAGE_BITMAP, IMAGE_CURSOR 和 IMAGE_ICON,分别对应新图片的格式,lParam 是图片的句柄,如果是位图,lParam 就是用 LoadBitmap 装入的位图句柄,同样,图片类型是光标和图标的时候,这里就是用 LoadCursor 和 LoadIcon 装入的句柄。

在例子程序中,用来改变图片框图片的语句是:

```
invoke SendDlgItemMessage, hWnd, IDC_BMP, STM_SETIMAGE, IMAGE_BITMAP, hBmp
```

hBmp 是用 LoadBitmap 装入的位图句柄, IDC_BMP 是图片框的 ID, wParam 参数用 IMAGE_BITMAP 表示要设置的图片类型是位图。

5. 使用文本编辑控件

文本编辑控件是基于 Edit 类的控件,可以用缩写 EDITTEXT 定义,读者可以在文本编辑控件中输入并编辑文本。每当用户在文本编辑控件中输入一个字符的时候,控件就会向对话框过程发送一个 WM_COMMAND 消息,所以在例子程序中,当在自定义文字的编辑框中每输入一个字,标题栏文字就会马上改变。

要获取编辑框中的文本有多种方法,可以用 GetWindowText,也可以用发送 WM_GETTEXT 消息的办法,要设置文本,同样可以用 SetWindowText 或发送 WM_SETTEXT,但最简便的办法还是使用下面的函数:

```
invoke  GetDlgItemText, hDlg, nIDDlgItem, lpString, nMaxCount    ;取文本
invoke  SetDlgItemText, hDlg, nIDDlgItem, lpString              ;设置文本
```

lpString 是放置字符的缓冲区地址,用 GetDlgItemText 函数来获取文本的时候,要用 nMaxCount 参数指定缓冲区的最大长度,以免获取的文本长度超过缓冲区长度引起溢出,设置的时候若使用 SetDlgItemText 函数时就不需要这个参数。

在实际使用中,经常要在文本编辑控件中输入输出数值型参数,将文本转换为数值比较麻烦,把数值转换为文本也要经过一个 sprintf 调用,为了简化操作,Windows 提供了两个函数来处理这个问题:

```
invoke  SetDlgItemInt, hDlg, nIDDlgItem, uValue, bSigned        ;设置控件中的数值
invoke  GetDlgItemInt, hDlg, nIDDlgItem, lpTranslated, bSigned  ;取控件中的数值
```

SetDlgItemInt 函数将 uValue 参数先转换成字符串格式,然后设置到文本编辑控件中, bSigned 参数指定了 uValue 的格式,如果是 TRUE 的话,表示 uValue 是符号数;是 FALSE 的话,表示 uValue 是无符号数。

GetDlgItemInt 函数则将对话框中的文本转换成数值型返回,同样,用 bSigned 指定转换的方式,TRUE 表示按照符号数格式转换,这时函数会检测文本的第一个字符是不是负

号；FALSE 则按照无符号数转换。参数 lpTranslated 是指向一个 dword 型变量的指针，GetDlgItemInt 会在这个变量中返回 BOOL 类型值表示函数是否调用成功，成功则返回 TRUE，有这样一个参数的原因是函数的返回值用来返回转换后的数值了，以至于没有地方可以表示函数是否执行成功。当然，lpTranslated 参数也可以输入 NULL，这样，当函数返回 0 的时候就无法知道是文本框是“0”还是文本不符合格式造成转换失败。

SetDlgItemInt 和 GetDlgItemInt 函数不仅适用于文本编辑控件，所有对其上面的文本可以修改的控件都可以使用它们。

使用文本编辑控件的时候，文本的长度也是个需要注意的问题。如果控件的宽度定义得过窄，当字符填充到最右边的时候，编辑框就不允许继续输入了，为了继续输入并让文本自动卷动，可以指定 WS_HSCROLL 风格；反之，定义 WS_HSCROLL 风格后输入文本的长度不受限制又不好，那么可以用向控件发送 EM_LIMITTEXT 消息的方法来设定最大长度。下面的例子将 IDC_EDIT 的输入最大长度定为 10 个字符：

```
invoke SendDlgItemMessage, hDlg, IDC_EDIT, EM_LIMITTEXT, 10, NULL
```

另外，有时候可能需要把编辑框设置为只读的（和灰化不同，灰化的编辑框中文本无法进行任何操作，包括卷动操作，而只读的仅仅是不能修改），要把初始状态定义为只读的，只需在定义语句中加上 ES_READONLY 风格，在程序中需要动态改变只读状态可以发送 EM_SETREADONLY 消息，下面的第一句把编辑框设为只读，第二句把编辑框改回到可写状态：

```
invoke SendDlgItemMessage, hDlg, IDC_EDIT, EM_SETREADONLY, TRUE, NULL ;只读
invoke SendDlgItemMessage, hDlg, IDC_EDIT, EM_SETREADONLY, FALSE, NULL ;可写
```

文本编辑框在默认状态下是单行的，也可以通过加上 ES_MULTILINE 风格变成多行的，这时可以同时加上 WS_VSCROLL 风格显示一个垂直方向的滚动条。

6. 使用滚动条

滚动条有水平和垂直两种，默认的 SCROLLBAR 语句定义的是水平的滚动条，它的默认风格是 SBS_HORZ，例子程序中用下面的语句定义了一个水平滚动条：

```
SCROLLBAR IDC_SCROLL, 6, 118, 125, 10
```

如果要定义垂直的滚动条，那么要指明 SBS_VERT 风格：

```
SCROLLBAR IDC_SCROLL, x, y, 宽度, 高度, SBS_VERT
```

和其他子窗口控件发送 WM_COMMAND 消息不同，水平滚动条向对话框窗口发送 WM_HSCROLL 消息，而垂直滚动条则发送 WM_VSCROLL 消息，所以针对两种方式的滚动条要分别处理不同的消息。

WM_xSCROLL 消息的参数如下所示：

```
wParam 的低 16 位 = nScrollCode ;动作码
wParam 的高 16 位 = nPos ;滚动条当前位置
lParam = hwndScrollBar ;滚动条控件的窗口句柄
```

其中 nScrollCode 代表了滚动条的当前动作，定义值及其含义如下：

- SB_BOTTOM——滚动条移到了最下/右边。
- SB_ENDSCROLL——用户停止了滚动动作。
- SB_THUMBPOSITION——滚动条被拖动到某处。
- SB_THUMBTRACK——滚动条在拖动中。
- SB_TOP——滚动条移到了最上/左边。
- SB_LINELEFT——滚动条左移了一格（对于水平滚动条）。
- SB_LINERIGHT——滚动条右移了一格（对于水平滚动条）。
- SB_PAGELEFT——滚动条左移了一页（对于水平滚动条）。
- SB_PAGERIGHT——滚动条右移了一页（对于水平滚动条）。
- SB_LINEDOWN——滚动条下移了一格（对于垂直滚动条）。
- SB_LINEUP——滚动条上移了一格（对于垂直滚动条）。
- SB_PAGEDOWN——滚动条下移了一页（对于垂直滚动条）。
- SB_PAGEUP——滚动条上移了一页（对于垂直滚动条）。

nPos 的值只有当动作码是 SB_THUMBPOSITION 或 SB_THUMBTRACK 时才有效，其他的时候为 0，图 5.7 示出了鼠标点击滚动条各处时对应的 nScrollCode。

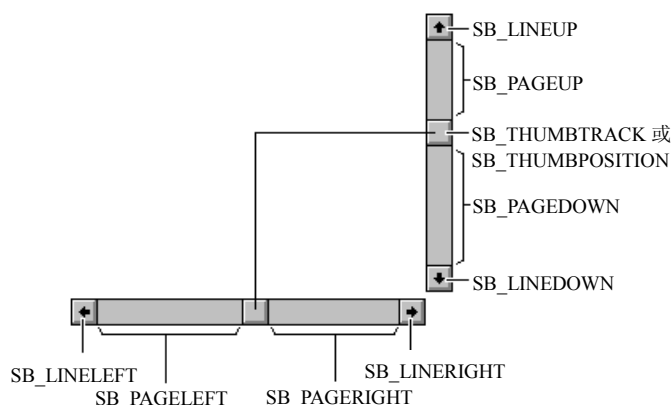


图 5.7 鼠标点击滚动条各处时产生的 nScrollCode

第一眼看到 SB_xxx 动作码的时候，读者可能会以为水平滚动条和垂直滚动条的动作码是不相同的——水平滚动条是 SB_xxxLEFT、SB_xxxRIGHT，而垂直滚动条是 SB_xxxUP、SB_xxxDOWN，但在 Windows.inc 中查看一下就可以发现，SB_xxxLEFT 和 SB_xxxUP 在数值上是相同的，SB_xxxRIGHT 和 SB_xxxDOWN 也是如此，所以不同定义方法只是为了直观起见而已。

以水平滚动条为例，处理滚动条消息的代码一般是如下结构：

```
.elseif eax == WM_HSCROLL ;窗口的消息处理分支，eax 为 wParam
    mov     eax, lParam
```

```

        .if      eax == hWnd 滚动条 1
        mov     eax, wParam
        .if      ax ==      SB_LINELEFT
            dec     位置变量
        .elseif  ax ==      SB_LINERIGHT
            inc     位置变量
        .elseif  ax ==      SB_PAGELEFT
            sub     位置变量, 页长
        .elseif  ax ==      SB_PAGERIGHT
            add     位置变量, 页长
        .elseif  ax ==      SB_THUMBPOSITION || ax == SB_THUMBTRACK
            mov     eax, wParam
            shr     eax, 16
            mov     位置变量, eax
        .endif
    .elseif  eax == hWnd 滚动条 2
        ;处理滚动条 2 的代码, 同上面的结构
        ...
    .endif

```

由于在例子程序 Control.asm 中只定义了一个滚动条,所有的消息肯定都是它发出的,所以去掉了判断 lParam 是哪个滚动条的步骤直接处理 wParam 中的动作码。

在用户按动滚动条后,滚动条不会自己移动位置,它只是将用户的动作以 WM_xSCROLL 消息的形式反馈给程序,真正要移动它还是要靠程序来设置,所以代码中要根据不同的动作首先计算新的位置,并判断新的位置是否越界,例子程序中的这些代码判断新的位置是否超出 0~100 的范围,如果是,则校正到 0~100 之间:

```

    cmp     dwPos, 0
    jge     @F
    mov     dwPos, 0
@@:
    cmp     dwPos, 100
    jle     @F
    mov     dwPos, 100

```

在介绍 MASM 语句的时候提到过,因为 .if dwPos > 0 语句只可以用来比较无符号数,所以在这里使用 cmp 指令自己构建测试分支而不是使用 .if 伪指令。

当计算好新位置的时候要将位置设置回去,用户才会看到滚动条移动了,方法是向滚动条发送 SBM_SETPOS 消息:

```

invoke    SendDlgItemMessage, hWnd, IDC_SCROLL, SBM_SETPOS, dwPos, TRUE

```

最后一个参数为 TRUE 表示设置后重新绘画滚动条。

在初始化的时候,要给滚动条发送 SBM_SETRANGE 消息来设定滚动范围:

```

invoke    SendDlgItemMessage, hWnd, IDC_SCROLL, SBM_SETRANGE, 最小值, 最大值

```

如果需要获取滚动条的信息,可以尝试发送下面两个消息: SBM_GETPOS 可以获取滚动条的当前位置,也就是上一次用 SBM_SETPOS 设置的值; SBM_GETRANGE 可以获取滚动的范

围，也就是用 SBM_SETRANGE 设置的值。

7. 使用组合框

顾名思义，组合框是一个“组合”起来的東西，它由一个可供选择的列表和一个可供输入的 edit 类组合而成。组合框让用户既可以自己输入文本，也可以选择列表中的某一项当做输入。用不同的风格定义可以产生 3 种类型的组合框，如图 5.8 所示。左边的是 CBS_SIMPLE 风格的组合框，它的上面可以输入文本，下面的列表可供选择预设文本；中间的是 CBS_DROPDOWN 风格的组合框，上面同样可以输入文本，但下面的列表是下拉式的，平时处于收起状态，点击编辑框右边的三角形才会拉下来；右边的是 CBS_DROPDOWNLIST 风格的组合框，它仅是一个下拉的选择框，上面的框中不允许输入文字。

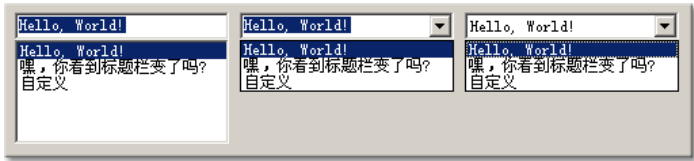


图 5.8 组合框的 3 种风格

组合框中还有几种常用的、可以附加的风格：

- CBS_AUTOHSCROLL——输入过长的文本时输入框自动滚动。
- CBS_LOWERCASE——自动将所有的文本转换成小写。
- CBS_SORT——自动将插入的文本项排序。
- CBS_UPPERCASE——自动将所有的文本转换成大写。

组合框中列表框部分的文字添加、项目的选择等操作都是通过发送消息来完成的，主要的消息如表 5.5 所示。

表 5.5 组合框的消息

消 息	wParam	lParam	说 明
CB_ADDSTRING	0	字符串地址	把一个字符串添加到列表中
CB_INSERTSTRING	位置索引	字符串地址	把一个字符串插入到列表中
CB_FINDSTRING	开始查找的位置索引	查找的字符串	在列表中查找以 lParam 字符串开头的项，找到则返回位置索引，未找到则返回 CB_ERR
CB_FINDSTRINGEXACT	位置索引	查找的字符串	精确查找字符串
CB_DELETESTRING	位置索引	0	删除一个列表项
CB_RESETCONTENT	0	0	删除所有的列表项
CB_GETLBTEXT	位置索引	缓冲区地址	获取指定列表项的字符串，缓冲区必须足够大
CB_GETLBTEXTLEN	位置索引	0	获取指定列表项的字符串长度
CB_GETCOUNT	0	0	获取列表项的总项数

续表

消 息	wParam	lParam	说 明
CB_SETCURSEL	位置索引	0	选中一个列表项，并将列表项中的文字拷贝到编辑控件中
CB_SELECTSTRING	开始查找的位置索引	字符串地址	查找以 lParam 指定的字符串开始的列表项，如果找到则选中它并将字符串拷贝到编辑控件中
CB_GETCURSEL	0	0	获取当前选中的位置索引，没有选中的项目则返回 CB_ERR
CB_SHOWDROPDOWN	状态	0	打开（状态为 TRUE）或收起（状态为 FALSE）下拉列表
CB_GETDROPPEDSTATE	0	0	检测列表的当前下拉状态，返回 TRUE 表示拉下，FALSE 表示收起

当用户在组合框中进行选择操作时，Windows 向对话框过程发送 WM_COMMAND 消息，消息中 wParam 参数的低 16 位是组合框 ID，高 16 位是通知码，用来表示用户的操作，通知码的定义如表 5.6 所示。

表 5.6 用户操作组合框后的通知码

通 知 码	说 明
CBN_SELCHANGE	用户将要选择一个项目（鼠标移动到了这个项目上）
CBN_CLOSEUP	下拉列表关闭（可能是选择完成也可能是取消选择）
CBN_SELENDOK	用户完成选择项目
CBN_SELENCANCEL	用户取消选择（鼠标移动到了某个项目上，但并没有按下而是点击了其他控件，或按了 Esc 键）
CBN_DBLCLK	在 CBS_SIMPLE 的组合框中双击了一个列表项
CBN_DROPDOWN	用户打开了下拉框（按动了编辑框边的下拉按钮）

如果想在用户选择了一个项目后做相应的动作，最好的办法就是处理 CBN_SELENDOK 通知码，因为这才意味着用户真正完成了一个选择动作，例子程序中就是这样处理的：

<pre>.elseif ax == IDC_TITLETEXT ;在 WM_COMMAND 消息中 shr eax, 16 .if ax == CBN_SELENDOK invoke SendDlgItemMessage, hWnd, IDC_TITLETEXT, \ CB_GETCURSEL, 0, 0 ;根据返回的 eax 值做相应动作... .endif</pre>
--

以上的操作都是针对下拉列表部分的，另外也有很多消息是针对组合框中的编辑控件的，对组合框的窗口句柄发送 WM_GETTEXT 和 WM_SETTEXT，操作的对象就是组合框的编辑控件；如果要限制编辑控件中文本的最大输入长度，可以发送 CB_LIMITTEXT 消息，这时候 wParam 参数指定最大数量；当用户在编辑框中编辑文本的时候，Windows 在用户输入之后、字符显示之前会发送 CBN_EDITUPDATE 通知码；当字符在编辑框中显示以后，又会发送 CBN_EDITCHANGE 通知码。所以在处理 WM_COMMAND 消息时通过处理这两个通知码可以检测到用户的输入操作。

哭

[illegible]

定义列表框时可以使用的风格如表 5.7 所示。

表 5.7 列表框可以使用的风格

风 格	说 明
LBS_DISABLENOSCROLL	在不需滚动的时候也显示垂直滚动条
LBS_EXTENDESEL	在多选列表框中允许按住 Shift 键的同时选中一个范围
LBS_MULTIPLESEL	允许多选，如果不定义的话则是单选列表框
LBS_NOSEL	列表框项目只能查看不能选择
LBS_NOTIFY	用户单击或双击项目时向父窗口发送 WM_COMMAND 消息
LBS_SORT	自动按字母顺序排序插入的项目
LBS_USETABSTOPS	列表框项目的文本中允许将 Tab 字符的位置展开
LBS_STANDARD	组合 LBS_NOTIFY, LBS_SORT, WS_VSCROLL 和 WS_BORDER

一般单选列表框只需定义 LBS STANDARD 就可以了。

汇编源代码 Listbox.asm 如下所示:

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include        windows.inc
include        user32.inc
includelib     user32.lib
include        kernel32.inc
includelib     kernel32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Equ 等值定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN      equ        1000h
DLG_MAIN      equ        1
IDC_LISTBOX1  equ        101
IDC_LISTBOX2  equ        102
IDC_SEL1      equ        103
IDC_RESET    equ        104
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .data?
hInstance dd        ?
        .const
szText1      db        ' 项目 1',0
szText2      db        ' 项目 2',0
szText3      db        ' 项目 3',0
szPath       db        ' *.*',0
szMessage    db        ' 选择结果: %s',0
szTitle      db        ' 您的选择',0
szSelect     db        ' 您选择了以下的项目: '
szReturn     db        0dh,0ah,0

```

器器

172

下面结合源程序来说明列表框的使用。当列表框有 LBS_NOTIFY 风格的时候，用户有所动作时列表框会向父窗口发送 WM_COMMAND 消息，同时在 wParam 的高 16 位中指定通知码，列表框的通知码种类很少，基本上就是以下几种：

- LBN_DBLCLK——用户双击了一个项目。
- LBN_ERRSPACE——插入项目时无法申请到足够的内存。
- LBN_KILLFOCUS——输入焦点被切换到其他控件中，列表框丢失了焦点。
- LBN_SELCANCEL——用户撤销了一个选择。
- LBN_SELCHANGE——选定状态改变。
- LBN_SETFOCUS——列表框得到输入焦点。

我们最关心的是 LBN_DBLCLK 和 LBN_SELCHANGE 通知码，在单选列表框中，如果程序用双击来选择项目，那么就要处理 LBN_DBLCLK 通知，例子程序中当用户双击 IDC_LISTBOX1 时弹出一个消息框，读者可以查看其使用方法。在多选列表框中，由于用户可能选择了多个项目，所以一般不用双击的方法选定；如果收到 LBN_SELCHANGE 通知的话，可以得知用户有一个选择动作，在这里可以进行相应的操作。

列表框通知父窗口是通过发送 WM_COMMAND 消息，而程序控制列表框的时候是通过向列表框发送消息来完成的，常用的列表框消息如表 5.8 所示。

表 5.8 列表框消息

消 息	wParam	lParam	说 明
LB_ADDSTRING	0	字符串地址	添加一个项目，返回加入后的索引
LB_DELETETEXT	位置索引	0	删除一个项目，返回剩余的项数
LB_FINDSTRING	开始索引	字符串地址	查找以字符串开头的项目，找到则返回位置索引，未找到则返回 LB_ERR
LB_FINDSTRINGEXACT	开始索引	字符串地址	精确查找一个项目，返回值同上
LB_GETANCHORINDEX	0	0	返回多选列表框多选时的起始位置
LB_GETCARETINDEX	0	0	多选列表框中的当前焦点项目位置
LB_GETCOUNT	0	0	返回列表框中的项目总数
LB_GETCURSEL	0	0	返回单选列表框当前选中的项目
LB_GETSEL	位置索引	0	检测指定项目的选中状态，返回非 0 为选中，返回 0 为未选中
LB_GETSELCOUNT	0	0	返回多选列表框选中项目的总数
LB_GETSELITEMS	最大项数	缓冲区地址	返回多选列表框的选中项目索引列表到缓冲区中
LB_GETTEXT	位置索引	缓冲区地址	返回某个项目的字符串
LB_GETTEXTLEN	位置索引	0	返回某个项目的字符串长度
LB_GETTOPINDEX	0	0	返回当前可见的第一个项目位置
LB_INSERTSTRING	插入位置	字符串地址	在指定位置插入一个项目
LB_RESETCONTENT	0	0	删除所有项目
LB_SELECTSTRING	开始位置	字符串地址	将以指定字符串开头的项目选中

续表

消 息	wParam	lParam	说 明
LB_SETCURSEL	位置索引	0	在单选列表框中选中一个项目
LB_SETSEL	选择状态	位置索引	在多选框中将一个项目选中或清除
LB_SETTOPINDEX	位置索引	0	滚动显示到指定的项目
LB_DIR	属性	文件通配符	搜索目录并将符合文件通配符的文件名加入到列表框中
LB_SEITEMRANGE	选择状态	范围	在多选框中将一个范围选中或清除

在这些消息中 LB_DIR 是个比较有趣的消息，它可以将指定目录中的文件名自动列出来并加入列表框中，如例子中用*. *将当前目录的全部文件名加到列表框中。LB_DIR 消息中 wParam 参数可以指定的属性可以是以下值的组合：

- DDL_ARCHIVE 加入归档属性的文件。
- DDL_DIRECTORY 加入目录。
- DDL_DRIVES 加入驱动器名。
- DDL_HIDDEN 包含隐含文件。
- DDL_READONLY 包含只读文件。
- DDL_READWRITE 包含可读写的文件。
- DDL_SYSTEM 包含系统文件。

在列表框中初始化时加入项目可以使用 LB_ADDSTRING 和 LB_INSERTSTRING 消息，删除项目可以用 LB_DELETESTRING 消息，删除全部项目用 LB_RESETCONTENT 消息。

对于单选列表框，要获取选中项目可以发送 LB_GETCURSEL 消息，要得到这个项目的字符串需要再用索引值通过 LB_GETTEXT 消息获取，读者可以查看例子中处理 LBN_DBLCLK 通知码的部分代码。

对于多选列表框，需要用 LB_GETSELITEMS 消息获取全部选中项目，这个消息返回的是一个列表，所有选中项目的索引按顺序排列返回到缓冲区中，所以在例子中处理“查看”按钮消息（IDOK）的时候，程序先发送 LB_GETSELCOUNT 消息得到选中的项目数，以便在下面用一个循环获取所有的项目，得到项目数后，再用 LB_GETSELITEMS 将选中项目的索引取到@szBuffer 中，接下来进入一个循环，循环的次数就是 LB_GETSELCOUNT 得到的数值，在循环中，程序从@szBuffer 中将索引值逐个取出并用 LB_GETTEXT 消息获取每一项的字符串，最后用一个 MessageBox 显示出来。

5.5 字符串资源

程序中用到的字符串常常定义在 .const 段中，但 Windows 也提供了另外一种使用字符串常量的方法，那就是在资源中定义。虽然在资源中定义字符串使用起来比直接在 .const 段中定义要复杂一点，但它带来的好处是便于开发不同语言的版本，比如，要推出其他语种的版

本只需要修改资源中的字符串表就可以了，即使语言转换的工作是由第三者通过修改可执行文件来做的（如编程爱好者常常做的汉化工作），修改资源也远比修改代码来得快捷和安全。

另外，有些 API 使用到的字符串必须定义在资源里面，如显示菜单帮助的 MenuHelp 函数等。

在资源脚本中定义字符串的语法是：

```

STRINGTABLE [DISCARDABLE]
BEGIN
    字符串 ID1 "字符串 1"
    字符串 ID2 "字符串 2"
    . . .
END

```

全部字符串组成一个字符串表，和其他资源定义不同，由于整个资源文件中只能定义一个字符串表，所以字符串表没有资源 ID，但是表中的不同字符串分别有一个字符串 ID。

在程序中使用字符串资源也很简单，用 LoadString 把字符串装入到缓冲区中去就可以用了：

```

invoke    LoadString, hInstance, 字符串 ID, addr 缓冲区, sizeof 缓冲区

```

为了防止溢出，最后一个参数指定缓冲区的长度。

如果要在单个可执行文件中实现多语种，那么可以在字符串表中定义不同语言的字符串。同一语种的字符串按规律排列，如下列中文的以 1000 开头，英文的以 2000 开头：

```

stringtable
{
    1001    "文件未找到！"
    1002    "无法打开文件！"
    ...
    2001    "File not found!"
    2002    "Can not open file!"
    ...
}

```

在程序中使用的时候，先确定一种语言并预先设置在 dwLanguage 变量中，使用中文时将 dwLanguage 设置为 1000，使用英文时设置为 2000，再写一个读取不同版本字符串的子程序_GetString，这样调用_GetString 子程序后就不用考虑版本问题了：

```

_GetString    proc _dwID, _lpBuffer, _dwSize

                pushad
                mov     eax, _dwID
                add     eax, dwLanguage
                invoke   LoadString, hInstance, eax, _lpBuffer, _dwSize
                popad
                ret

_GetString    endp

```

5.6 版本信息资源

有时应用程序需要确保自己运行时使用某一特定版本的 DLL，以便确保可以使用某些函数。检测版本是通过 API 函数查询定义于资源中的版本信息来完成的，如果资源中没有定义版本，那么就无法知道一个文件的版本究竟是多少。

版本信息是以 VERSIONINFO 类型的资源保存在应用程序中的，里面可以定义的信息包括文件的版本号、创建单位和语种等。版本信息的定义是可选的，一个程序可以不定义版本信息资源，如果定义了的话，也不一定要定义全部信息项目。

如果一个文件定义有版本信息资源，那么在文件的属性页（在文件图标上按鼠标右键，在弹出的菜单上选择）上就会有一个“版本”页面，如图 5.10 所示。

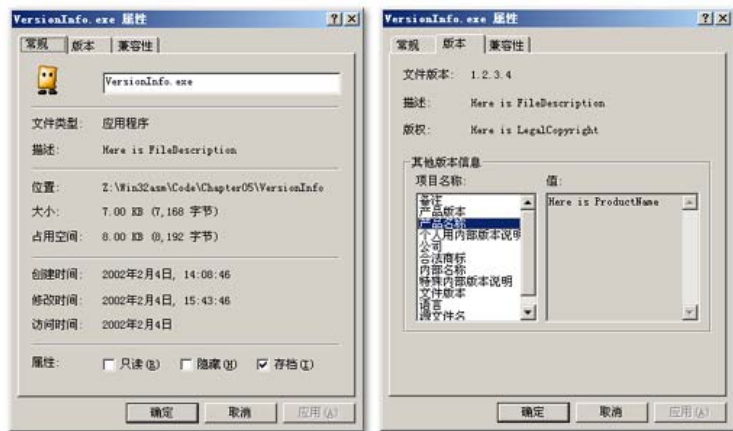


图 5.10 文件属性中的版本信息

5.6.1 版本信息资源的定义

在所附光盘的 Chapter05\VersionInfo 目录中有一个例子，在该目录的 Version.rc 文件中定义了一个版本信息，读者可以看编译后的 VersionInfo.exe 中的“版本”属性页，对比一下资源定义中的内容究竟出现在属性页的哪些地方，定义的代码如下：

```
1 VERSIONINFO
FILEVERSION 1, 2, 3, 4
PRODUCTVERSION 2, 3, 4, 5
FILEOS VOS_WINDOWS32
FILETYPE VFT_APP
FILESUBTYPE VFT2_UNKNOWN
BEGIN
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x804, 0x4b0
    END
    BLOCK "StringFileInfo"
```



```

BEGIN
    BLOCK "080404b0"
    BEGIN
        VALUE "Comments", "Here is Comments\0"
        VALUE "CompanyName", "Here is CompanyName\0"
        VALUE "FileDescription", "Here is FileDescription\0"
        VALUE "FileVersion", "1, 0, 0, 1\0"
        VALUE "InternalName", "Here is InternalName\0"
        VALUE "LegalCopyright", "Here is LegalCopyright\0"
        VALUE "LegalTrademarks", "Here is LegalTrademarks\0"
        VALUE "OriginalFilename", "Here is OriginalFilename\0"
        VALUE "PrivateBuild", "Here is PrivateBuild\0"
        VALUE "ProductName", "Here is ProductName\0"
        VALUE "ProductVersion", "1, 0, 0, 1\0"
        VALUE "SpecialBuild", "Here is SpecialBuild\0"
    END
END
END

```

现在来看这些定义语句的含义。首先，版本信息定义的语句格式是：

```

版本信息资源 ID  VERSIONINFO
固定属性
BEGIN
    块声明定义
    ...
END

```

版本信息资源 ID 的取值必须为 1，如果不为 1 则属性页上的“版本”信息是无法显示出来的。（笔者也不明白为什么必须为 1 还要定义这个 ID，像 stringtable 一样没有 ID 不就完事了？）

可以定义的固定属性有：

- FILEVERSION——定义文件版本号，可以定义 4 个 16 位版本号 xx.xx.xx.xx。
- PRODUCTVERSION——定义产品版本号，可以定义 4 个 16 位版本号 xx.xx.xx.xx。
- FILEFLASMASK——指定 FILEFLAGS 属性中哪些位有效。
- FILEFLAGS——文件标志，是一些标志位的组合：VS_FF_PATCHED，VS_FF_DEBUG，VS_FF_PRIVATEBUILD，VS_FF_INFOINFERRED，VS_FF_PRERELEASE 和 VS_FF_SPECIALBUILD。
- FILEOS——定义适用的操作系统，可以定义为 VOS_UNKNOWN，VOS_DOS，VOS_NT，VOS_WINDOWS16，VOS_WINDOWS32，VOS_DOS_WINDOWS16，VOS_DOS_WINDOWS32 或 VOS_NT_WINDOWS32。
- FILETYPE——定义文件类型，可以是 VFT_UNKNOWN，VFT_APP，VFT_DLL，VFT_DRV，VFT_FONT，VFT_VXD 或 VFT_STATIC_LIB。
- FILESUBTYPE——定义文件的子类型。当文件类型是 VFT_DRV（驱动程序）的时候，这里可以是 VFT2_UNKNOWN，VFT2_DRV_COMM，VFT2_DRV_PRINTER，

VFT2_DRV_KEYBOARD, VFT2_DRV_LANGUAGE, VFT2_DRV_DISPLAY, VFT2_DRV_MOUSE, VFT2_DRV_NETWORK, VFT2_DRV_SYSTEM, VFT2_DRV_INSTALLABLE 或 VFT2_DRV_SOUND; 当文件类型是 VFT_FONT (字体) 的时候, 这里可以是 VFT2_UNKNOWN, VFT2_FONT_RASTER, VFT2_FONT_VECTOR 或 VFT2_FONT_TRUETYPE。

在固定属性定义完成以后, 需要定义一些块声明, 块声明有两种: 变量型的信息块和字符串类型的信息块, 变量类型的信息块定义如下:

```
BLOCK "VarFileInfo"
BEGIN
    VALUE "Translation", 语言 ID, 字符集 ID
    .....
END
```

语言 ID 的常用值有 0x0404 (繁体中文)、0x0409 (美国英语) 和 0x0804 (简体中文), 字符集 ID 的常用值有 0 (7 位 ASCII)、950 (台湾 GB5) 和 1200 (Unicode)。一般使用 0x804, 0x4b0 来定义, 也就是简体中文和 Unicode (0x4b0=1200)。其他还有很多取值, 读者可以查看 Rc.exe 的帮助文件。

变量类型信息块用来表示 VERSIONINFO 资源中定义有哪些语言和字符集的字符串类型信息块。如上例中有一句 VALUE "Translation", 0x804, 0x4b0 表示对应有一个名为 "080404b0" 的字符串类型的信息块。

字符串信息块的定义语句为:

```
BLOCK "StringFileInfo"
BEGIN
    BLOCK "语言集"
    BEGIN
        VALUE "字符串名称", "字符串"
        .....
    END
END
```

语言集就是变量类型中定义的, 其名称一定要是将语言 ID 和字符集 ID 组合成一个 8 位的十六进制的格式, 以上例文件来说明, 当变量类型的信息块种定义 0x804, 0x4b0 时, 语言集名称就是 "080404b0", 在语言集块的定义中, 还可以定义多条字符串型的版本信息, 这些版本信息的字符串名有 12 种, 如表 5.9 所示。

表 5.9 版本信息字符串类型

字符串名称	属性页位置	说 明
Comments	备注	有关程序的附加说明信息
CompanyName	公司	开发产品的公司
FileDescription	描述	有关文件的简单描述
FileVersion	文件版本	如 1.50、5.0.RC2 等字符串型的版本信息
InternalName	内部名称	

续表

字符串名称	属性页位置	说 明
LegalCopyright	版权	文件的所有版权信息
LegalTrademarks	合法商标	文件的所有注册商标信息
OriginalFilename	源文件名	原始文件名，从这里可以得知文件是否被改名
PrivateBuild	个人用内部版本说明	作者私人信息
ProductName	产品名称	文件所属的产品名称
ProductVersion	产品版本	文件所属的产品的版本号
SpecialBuild	特殊内部版本说明	特殊说明

定义版本信息字符串的时候要注意，所有的字符串必须是以 NULL 结尾的串，所以要在字符串尾加上\0，如例子程序所示：

```
VALUE "Comments", "Here is Comments\0"
VALUE "CompanyName", "Here is CompanyName\0"
...
```

5.6.2 在程序中检测版本信息

Win32 API 中有 3 个版本信息函数：GetFileVersionSize, GetFileVersionInfo 和 VerQueryValue，它们驻留在 VERSION.DLL 文件中，如果在源程序中使用它们，注意要加上 include Version.inc 以及 includelib Version.lib 语句。

用这 3 个函数获取版本信息的方法是：

首先调用 GetFileVersionInfoSize 函数检测文件中有没有版本信息资源：

```
invoke GetFileVersionInfoSize, addr szFile, NULL
```

其中 szFile 是要检测的 PE 文件名字符串，该函数的返回值是版本信息资源的长度，如果返回 0，则表示文件不是 PE 文件或没有定义版本信息资源。

如果检测到文件中有版本信息资源，那么可以将版本信息资源读取到一个缓冲区中，缓冲区的长度必须足够容纳上一步返回的资源长度，方法是：

```
invoke GetFileVersionInfo, addr szFile, NULL, \
      sizeof dbVerInfo, addr dbVerInfo
```

其中 dbVerInfo 为一个足够大的缓冲区。该函数会把整个版本信息资源拷贝到这个缓冲区中。

拷贝到缓冲区中的信息有它自己的格式，必须用 VerQueryValue 去“解码”，解码固定属性的方法是：

```
invoke VerQueryValue, addr dbVerInfo, addr szRoot, addr lpBuffer, addr dwLen
```

第一个参数指向前一步返回的版本信息资源数据，第二个参数指向一个字符串：“\”，第三个和第四个参数指向 dw 类型的变量 lpBuffer 和 dwLen，返回到 lpBuffer 中的是指向一个 VS_FIXEDFILEINFO 结构的指针，这个结构中有定义的固定属性内容。

如果要获取字符串类型信息块中的版本信息，那就比较复杂一点了，必须首先知道语

言集的名称，所以先要获取版本信息资源中变量类型信息块的内容，方法是：

```

invoke    VerQueryValue, addr dbVerInfo, addr szVarInfo, \
          addr lpBuffer, addr dwLen
mov       eax, lpBuffer
mov       eax, [eax]
ror       eax, 16

```

szVarInfo 是一个字符串：“\VarFileInfo\Translation”，这时函数在 lpBuffer 中返回语言集变量的指针，所以要先 mov eax, lpBuffer，再用 eax 做指针用 mov eax, [eax] 得到语言集变量。语言集变量的高 16 位是字符集 ID，低 16 位是语言 ID，可以使用 ror eax, 16 位来调换高低位，以我们的例子为例，现在 eax 中的值就是 080404b0h 了！

接下来就可以获取字符串版本信息了，先将语言集的值通过 wsprintf 函数转换成“080404b0”的形式，然后拼装成“\StringFileInfo\080404b0\字符串名称”形式的字符串，中间的“字符串名称”可以是表 5.9 中的 12 种名称之一，最后调用下面的语句（假定拼装好的字符串地址为 szString）：

```

invoke    VerQueryValue, addr dbVerInfo, addr szString, addr lpBuffer, addr dwLen

```

执行后 lpBuffer 中会得到一个指针，指向版本信息字符串定义的内容，这就是我们最后需要的结果！重复这个步骤可以得到所有 12 种字符串版本信息。

读者可以在所附光盘的 Chapter05\ShowVersionInfo 目录中找到一个 ShowInfo 程序，它可以获取 PE 文件中的版本信息资源并显示出来，详细的代码请参考该目录中的文件，主文件 ShowInfo.asm 是界面程序，版本信息资源的代码在 GetVersionInfo.inc 文件中，由于篇幅有限，源程序在这里就不列出来了。

5.7 二进制资源和自定义资源

5.7.1 使用二进制资源

在第 2 章中曾经提到 DOS 的 exe 文件可以带一个覆盖部分，覆盖部分实际上就是在真正的可执行部分后面附加的数据，然后由程序在运行中打开自身文件并使用这些数据。Win32 的可执行文件中除了上面介绍的这些标准类型的资源外，也可以在程序中附带其他数据，当然方法完全不同——Win32 资源中允许用户自己定义二进制的资源或者自定义格式的资源，资源的内容可以是任何数据，也可以将一个磁盘文件按二进制格式包括进去。

二进制资源的定义格式是：

```

资源 ID    RCDATA [DISCARDABLE]
BEGIN
           数据定义
           ...
END

```

也可以用一个磁盘文件当做资源的内容：

资源 ID	RCDATA	[DISCARDABLE]	文件名
-------	--------	---------------	-----

在程序中要使用资源的内容时，可以通过以下步骤将资源装入内存使用：

- (1) 用 FindResource (hInstance, lpName, lpType) 查找资源。lpName 的值为资源 ID, lpType 的值为 RT_RCDATA，如果找到资源。那么函数返回一个资源信息句柄。
- (2) 用 LoadResource (hInstance, hResInfo) 装入资源。hResInfo 是上一步中得到的资源信息句柄，装入成功的话函数会返回一个资源句柄。
- (3) 用 LockResource (hResData) 将资源锁定到内存中。hResData 是上一步得到的资源句柄，函数返回资源装入的内存地址，程序就可以使用内存中的数据了。
- (4) 如果想知道装入资源的大小是多少，可以使用 FindResource 返回的 hResInfo 来调用 SizeofResource (hInstance, hResInfo) 从而得到资源大小。

下面是一个装入资源 ID 为 ID_MYRES 的 RCDATA 类型资源的例子：

invoke	FindResource , hInstance, ID_MYRES, RT_RCDATA	;寻找资源
.if	eax	
	mov hResInfo, eax	
	invoke SizeofResource , hInstance, eax	;获取资源尺寸
	mov dwResSize, eax	
	invoke LoadResource , hInstance, hResInfo	;装入资源
	.if	eax
		invoke LockResource , eax ;锁定资源
		.if
		eax
		mov lpRes, eax
		;处理 lpRes 指向的资源内容
		.endif
		.endif
		.endif

5.7.2 使用自定义资源

自定义资源的定义格式比二进制资源更灵活，它和二进制资源的区别在于可以指定资源类别为自定义的名称：

资源 ID	类型 ID	[DISCARDABLE]
BEGIN	数据定义	
	...	
END		

或用一个磁盘文件当做资源的内容：

资源 ID	类型 ID	[DISCARDABLE]	文件名
-------	-------	---------------	-----

类型 ID 可以是大于 255 的数值（255 及以下的数值由 Windows 使用）或字符串，如可以定义如下：

1000	WAVE	"Hello. wav"	;定义类型为“WAVE”，资源 ID 为 1000 的资源
1000	TEXT	"Readme. txt"	;定义类型为“TEXT”，资源 ID 为 1000 的资源
1000	1000	"Test. bin"	;定义类型 ID 为 1000, 资源 ID 为 1000 的资源

在程序中使用自定义资源的方法和使用二进制资源类似，惟一的区别是使用 FindResource 得到 hResInfo 的参数有些区别，得到 hResInfo 以后的步骤是一模一样的。针对上面 3 句定义，查找资源的方法可以是：

szResType1	db	"WAVE", 0	
szResType2	db	"TEXT", 0	
		...	
invoke	FindResource ,	hInstance, 1000, addr szResType1	; 针对上面第一句
invoke	FindResource ,	hInstance, 1000, addr szResType2	; 针对上面第二句
invoke	FindResource ,	hInstance, 1000, 1000	; 针对上面第三句

在使用完二进制或自定义资源以后，不必使用任何函数去释放它们，Windows 在程序退出的时候会自动将它们释放。

定时器 and Windows 时间

6.1 定时器

在 DOS 操作系统中要用到定时功能的时候一般有两种方法，一种是用一个空循环来延时；二是截获时钟中断，计算机的硬件时钟中断会以每 55 ms 一次的频率触发 8 号中断，而在默认的 int 08h 中断处理程序中有一句调用 int 1ch 的代码，所以截获 int 08h 或 int 1ch 都可以达到定时的要求。第一种方法的定时效果随计算机主频的不同可能会大不相同，相比之下，第二种方法更为常用。

在 Windows 操作系统下，用户程序不可能去截获时钟中断，所以操作系统用提供定时器的方法来满足用户的类似需求。

6.1.1 定时器简介

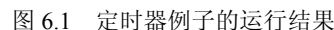
在应用程序需要使用定时器时，可以用 SetTimer 函数向 Windows 申请一个定时器，要求系统在指定的时间以后“通知”应用程序，如果申请成功的话，系统会以指定的时间周期调用 SetTimer 函数指定的回调函数，或者向指定的窗口过程发送 WM_TIMER 消息，与 DOS 操作系统固定以 55 ms 的间隔触发中断服务程序相比，SetTimer 函数可以指定的时间间隔更为灵活——以 ms 为单位，可以指定的时间周期为一个 32 位的整数，也就是从 1~4 294 967 295 ms，这可是一个将近 50 天的范围！

但是在具体的使用中不要被这个参数所迷惑：由于 Windows 的定时器同样是基于时钟中断的，所以虽然参数的单位是 ms，但精度还是 55 ms，如果指定一个小于 55 ms 的周期，不管是 1 ms 还是 54 ms，Windows 最快也只能在每个时钟中断的时候触发这个定时器，也就是说，实际上这个定时器是以 55 ms 为触发周期的；另外，当指定一个时间间隔的时候，Windows 会以与这个间隔最接近的 55 ms 的整数倍时间来触发定时器，假定建立一个周期为 1 000 ms 的定时器，定时器的触发周期实际上不是 1 s 而是 989 ms (55 ms×18)。

使用定时器时还有一个要点就是定时器消息是一个低级别的消息，这表现在两个方

由此可见,应用程序不能依靠定时器来保证某件事情必须在规定的时刻被处理,另外,也不能依赖对定时器消息计数来确定已经过去了多少时间。

这个例子程序中共定义了 3 个定时器，第 1 个以 250 ms 为周期更换对话框上的图标；第 2 个以 1s 为单位进行计数并把结果显示在对话框上；第 3 个以 2s 为单位驱动扬声器发出“嘟嘟”的响声。为了验证 WM_TIMER 消息的级别，读者可以在运行中按住标题栏的“关闭”按钮不放，就可以发现 3 个定时器全部停止了，然后将鼠标移出“关闭”按钮并释放，定时器会重新工作，但对话框上的计数结果在定时器停止的期间并没有补上去，也就是说，在这期间，WM_TIMER 消息被全部丢弃了。



下面以 Timer 程序为例说明定时器的使用方法，这个程序的资源脚本文件定义如下：

对资源的定义读者现在一定不会陌生了，这个文件中定义了两个图标和一个对话框，

器

[illegible]

这个程序的基本结构非常简单，就是一个标准的对话框程序而已，在 WM_INITDIALOG 中用 SetTimer 申请了 3 个定时器，并在 WM_CLOSE 消息中用 KillTimer 撤销这 3 个定时器。

申请一个定时器使用 `SetTimer` 函数，函数的使用方法如下：

```
invoke SetTimer,hWnd,nIDEvent,uElapse,lpTimerFunc
```

`hWnd` 参数是 `WM_TIMER` 消息发往的窗口句柄；`nIDEvent` 参数是一个用户指定的任意整数，用来标识一个程序中的多个定时器；`uElapse` 是时间周期，以 `ms` 为单位，这个参数是必须指定的；`lpTimerFunc` 是定时器过程，在下面的内容中有详细介绍。如果定时器建立成功的话，函数的返回值是定时器的标识符。

撤销定时器的函数是 `KillTimer`，该函数的使用方法是：

```
invoke KillTimer,hWnd,uIDEvent
```

参数 `hWnd` 和 `uIDEvent` 就是建立定时器时使用的数值。

使用 `SetTimer` 函数的方法有两种，第一种方法是要求 Windows 将 `WM_TIMER` 消息发往指定的窗口过程，这时候 `lpTimerFunc` 必须为 `NULL`，如例子中的：

```
invoke SetTimer,hWnd,ID_TIMER1,250,NULL          (例 1)
invoke SetTimer,hWnd,ID_TIMER2,2000,NULL
```

这两个句子设置了两个标识分别为 `ID_TIMER1` 和 `ID_TIMER2` 的定时器，定时周期分别为 `250 ms` 和 `2 s`。在窗口过程收到 `WM_TIMER` 消息的时候，`wParam` 中是用 `SetTimer` 建立定时器时使用的标识 `uIDEvent`，所以程序可以建立一个分支，通过判断 `wParam` 来处理不同的定时器引起的 `WM_TIMER` 消息。在例子中，当 `wParam` 是 `ID_TIMER1` 的时候更换图标框中的图标，是 `ID_TIMER2` 的时候用 `MessageBeep` 函数来发出一声“嘟”的声音。如果要撤销用这种方法建立的定时器，那么只需要用建立时的 `hWnd` 和 `uIDEvent` 参数简单地调用 `KillTimer` 就可以了。

还有一种使用定时器的方法，那就是要求 Windows 在时间到的时候调用指定的定时器过程，而不是某个窗口过程，那么只需要指定 `lpTimerFunc` 参数，如例子中的：

```
invoke SetTimer,NULL,NULL,1000,addr _ProcTimer    (例 2)
```

这句话要求系统把定时器消息发送到 `_ProcTimer` 定时器过程中去，但是，这时候没有参数用来指定定时器标识，到最后如何用 `KillTimer` 撤销这个定时器呢？答案是 `SetTimer` 函数会返回一个标识，程序可以保存这个标识并在 `KillTimer` 函数中使用。

当然，这种用法中的定时器标识也可以自己指定，但这时候一定要同时指定 `hWnd`，虽然这个 `hWnd` 没有实际的用途，如果 `hWnd` 为 `NULL`，那么即使指定了定时器标识，这个标识也会被忽略，如：

```
invoke SetTimer,hWnd,ID_TIMER3,1000,addr _ProcTimer    (例 3)
```

这个语句定义了一个标识为 `ID_TIMER3`、消息发往 `_ProcTimer` 子程序的定时器。

定时器过程是如下定义的：

```
TimerProc proc hwnd,uMsg,idEvent,dwTime
```

Windows 回调定时器过程的时候会有 4 个参数，`uMsg` 总是 `WM_TIMER`，`hwnd` 和

idEvent 是例 3 用法中指定的 hWnd 和定时器标识，如果是例 2 的用法，那么 hWnd 就是 NULL，而 idEvent 就是 SetTimer 返回的由 Windows 定义的定时器标识。由于有 idEvent 参数，所以我们同样可以把多个定时器消息指向同一个定时器过程中，并且根据 idEvent 参数构建一个分支来处理不同定时器引发的消息。

程序中还可能遇到一种情况：当在 SetTimer 中指定的定时器标识已经存在会怎样呢？答案是 Windows 会用新的参数代替老的定时器参数，函数执行以后，这个标识的定时器消息将以新的时间周期发送。



读者可能已经注意到，例子程序的窗口过程中把 WM_TIMER 的消息的处理代码放在第一个分支上，这是对程序的简单优化，把频繁发生的消息放到前面可以使程序少执行一系列的比较指令，像 WM_CREATE 和 WM_DESTROY 等仅发生一次的消息可以放到分支的最后面。

6.2 Windows 时间

很多读者看到“定时器”这个词的时候往往就联想到时钟，笔者也曾是如此，但是经过 6.1 节的介绍后就可以发现，定时器是不能用来构造时钟的，定时器用于时钟程序中只能是用在定时刷新屏幕这个功能上，要得到系统的时间还是要靠别的方法。同样道理，定时器也不能用于判断从上次定时器被触发后已经过去了多少时间。下面将介绍其他一些函数来完成这些功能。

6.2.1 Windows 时间的获取和设置

在 Win32 编程中，常用的获取系统时间的函数有 2 个：

```
invoke GetLocalTime,lpSystemTime
invoke GetSystemTime,lpSystemTime
```

它们之间的区别是：GetLocalTime 返回当前的时间，GetSystemTime 返回当前的格林威治标准时间，这两个函数返回的时间数据包括年、月、日、时、分、秒、毫秒，以及星期，由于数据比较多，所以无法放在 eax 中返回，应用程序需要预先设置一个 SYSTEMTIME 结构的缓冲区，并将缓冲区地址 lpSystemTime 当参数传递给函数，函数会把时间数据返回到这个缓冲区中。

SYSTEMTIME 结构的定义如下：

SYSTEMTIME	STRUCT	
wYear	WORD	? ;年
wMonth	WORD	? ;月
wDayOfWeek	WORD	? ;星期, 0=星期日, 1=星期一,
wDay	WORD	? ;日
wHour	WORD	? ;时
wMinute	WORD	? ;分
wSecond	WORD	? ;秒
wMilliseconds	WORD	? ;毫秒

SYSTEMTIME

ENDS

需要注意的是，结构中的字段全部是 word 类型的，而由于 Win32 程序中用的往往是 dword 型变量，所以在使用这些数据之前往往要先把它们转换为 dword 类型，用 movzx 指令就可以很方便地完成这个工作，如 movzx eax,stSystemTime.wYear 将 wYear 字段扩展到 32 位后放到 eax 中。

与获取系统时间的函数相对应，可以用下面的两个函数设置系统时间：

```
invoke SetLocalTime,lpSystemTime
invoke SetSystemTime,lpSystemTime
```

同样，SetLocalTime 中的参数代表本地时间，SetSystemTime 中的参数代表格林威治标准时间，在调用函数之前，要把需要设置的时间放到一个 SYSTEMTIME 结构中并把结构地址当做参数传递给 Windows。

6.2.2 计算时间间隔

在实际的编程中，经常要计算距离上次的时间点已经过去了多少时间，当然，这个数据可以通过两次调用 GetLocalTime 函数并将两次的时间值相减来得到，惟一的麻烦就是计算的过程比较复杂，因为 GetLocalTime 函数返回的数据中有年、月、日、时、分、秒和毫秒等数据，将两个时间相减要涉及借位的问题，似乎惟一合理的算法就是首先将这些数据合并成公元以来的总毫秒值再进行相减。

使用时间戳函数 GetTickCount 可以方便地完成这个功能，GetTickCount 函数返回的是 Windows 本次启动以来的 ms 数，得到的时间数值直接在 eax 中返回，这是一个 32 位的整数，可以表示的范围是 1~0xffffffff ms，所以当 Windows 连续运行 49.7 天以后，计数器会清零并重新开始。虽然从这个函数得到的计数值无法用来判断当前的具体时间，但是用来计算两个时间点之间的间隔是最方便不过的了，例如：

```
invoke GetTickCount
mov     dwTickCount1,eax
...
invoke  GetTickCount
sub     eax,dwTickCount1      ;现在 eax 中就是时间间隔的毫秒数
```

在 Windows 9x 系统下，GetTickCount 函数的精度为 55ms，任何两次调用 GetTickCount 函数后相减得到的时间间隔要么是 0，要么是 55ms 的整数倍。在 Windows NT/2000/XP 系统下，函数的精度是 10ms。

如果需要得到更精确的时间间隔值，那该怎么办呢？那就要用到 Windows 的高精度时间戳函数了。

Windows 在内部维护一个高精度的计时器，计时的精度取决于计算机的硬件速度，用 QueryPerformanceFrequency 函数可以获取该计时器每秒钟的计数值：

```
.data?
dqFreq dq ?
```

```
.code
invoke QueryPerformanceFrequency,addr dqFreq
```

由于计数值比较大，一个 32 位的整数无法容纳，所以该函数的参数指向一个 qword，函数运行后在该 qword 中返回一个 64 位的计数值，根据该计数值就可以算出计时器的精度为 $1\,000\,000 / dqFreq$ 微秒 (μs)。当 CPU 主频比较高的时候，计数值会比较大，意味着计时器的精度比较高。

得到精度值后，可以将该值保存下来，以使用来计算时间间隔。

QueryPerformanceCounter 函数可用来获取高精度计时器的计数值，该函数也是将 64 位的计数值返回到一个 qword 中。当两次调用函数得到的计数值是 X_2 和 X_1 ，而每秒计数值为 Y 时，时间间隔就是 $(X_2 - X_1) \times 1\,000\,000 / Y$ 微秒 (μs)。鉴于 64 位的除法在算法上比较复杂，在一般情况下可以用浮点指令来完成计算（对浮点指令不是很熟悉的读者可以参考 2.5.2 节中提及的《Intel Architecture Software Developer's Manual》或者其他相关书籍）：

```
.data
dqTickCount1 dq ? ;时间点 1 的计数值
dqTickCount2 dq ? ;时间点 2 的计数值
dqFreq dq ? ;计数精度
dqTime dq ? ;时间间隔
dw1m dd 1000000 ;常数

.code
invoke QueryPerformanceCounter,addr dqTickCount1 ;时间点 1
...
invoke QueryPerformanceCounter,addr dqTickCount2 ;时间点 2
invoke QueryPerformanceFrequency,addr dqFreq
mov eax,dword ptr dqTickCount1
mov edx,dword ptr dqTickCount1+4
sub dword ptr dqTickCount2,eax
sbb dword ptr dqTickCount2+4,edx
finit
fild dqFreq
fild dqTickCount2
fimul dw1m ;乘以 1000000
fdivr
fistp dqTime ;dqTime 中的 64 位值就是时间间隔（以微秒为单位）
```

第 7 章

图 形 操 作

图形设备接口 GDI (Graphics Device Interface) 是 Win32 的一个重要组成部分, 其作用是允许 Windows 的应用程序将图形输出到计算机屏幕、打印机或其他输出设备上。GDI 实际上是一个函数库, 包括直线、画图和字体处理等数百个函数。

7.1 GDI 原理

Windows 是基于图形界面的, 所以在 Win32 编程中, 图形操作是最常用的操作。GDI 的意义在于将程序对图形界面的操作和硬件设备隔绝开来, 在程序中可以将所有的图形设备都看成是虚拟设备, 包括视频显示器和打印机等, 然后通过 GDI 函数用同样的方法去操作它们, 由 Windows 负责将函数调用转化成针对具体硬件的操作。只要一个设备提供了和 Windows 兼容的驱动程序, 它就可以被看做是一个标准的设备。以前在 DOS 系统下写应用程序的时候, 如果要进行图形操作, 那么就要考虑到市场上每种显示卡的不同, 否则在装配某种显卡的计算机上就可能无法正常运行, 对汇编程序员来说, 这真是一个噩梦。在 Win32 编程中, 正是 GDI 函数让这个噩梦成为历史。

GDI 函数全部包括在 GDI32.DLL 中, 在编程的时候, 注意要在源程序的开头加上相应的包含语句:

```
include    gdi32.inc
includelib gdi32.lib
```

与 GDI 相关内容的规模真是太庞大了, 只要查看一下 gdi32.inc 文件就可以发现, 函数的总数达到了 300 多个, 与 GDI 相关的数据结构也非常多, 要完全深入 GDI 编程, 用上本书的全部篇幅可能也不够。在本章中, 笔者希望通过几个例子, 让读者能了解 GDI 的原理和基本的使用方法。

归纳起来, GDI 操作可以从 3 个方面去了解——When, Where 和 How:

- When——指的是进行图形操作的时机, 究竟什么时刻最适合程序进行图形操作呢? 在 7.1.1 节“GDI 程序的结构”中, 将探讨这个问题。

- Where——指的是图形该往哪里画，既然 Windows 隔离了硬件图形设备，那么该把什么地方当做“下笔”的地方呢？7.1.2 节的“设备环境”就是解答。
- How——了解了上面两个问题后，最后还要知道“如何画”，这就涉及如何使用大部分 GDI 函数的问题了，在本章余下来的篇幅中，将集中讨论这个问题。

7.1.1 GDI 程序的结构

1. 客户区的刷新

正如上面所说的，本节讨论的是“When”的问题，读者可能会问：为什么会有这个问题，如果要向窗口输出图形，程序想在什么时候输出那就是什么时候，难道这个时刻还有规定不成？

但这个问题似乎不能这样来问，让我们来考虑这些情况：在 DOS 操作系统中编程的时候，程序把文字或图形输出到屏幕，在输出新的内容之前，这些内容总是保留在屏幕原处，这些内容会被意外覆盖的惟一情况是激活一个 TSR 程序，但 TSR 程序在退出之前有义务恢复原来的屏幕，如果它无法恢复屏幕的内容，那么这是它的责任，我们不会在自己的程序中去考虑屏幕内容会无缘无故消失这种情况，所以可以把屏幕看成是应用程序私有的。

如果程序输出的内容过多，如用 dir 显示一个含有很多文件的目录，用户根本无法看清快速上翻的屏幕，这时程序可以设计一个参数来暂停一下，如 dir /p。这已经是 DOS 程序最“体贴”的做法了，如果用户想回过头去看已经滚出屏幕的内容，那可对不起，只能再执行一遍了！

所以对 DOS 程序来说，程序想在什么时候输出信息那就是什么时候，根本不存在 When 这个问题。

但在 Windows 操作系统中，屏幕是多个程序“公用”的，用户程序不要指望输出到窗口中的内容经过一段时间后还会保留在那里，它们可能被别的东西覆盖，如其他窗口、鼠标箭头或下拉的菜单等。在 Windows 中，恢复被覆盖内容的责任大部分属于用户程序自己，理由很简单：Windows 是个多任务的操作系统，假如程序 B 覆盖了程序 A 的窗口内容，覆盖掉的内容由程序 B 负责恢复的话，它就必须保存它覆盖掉的内容，但是在它将保存的内容恢复之前，程序 A 也在运行，并可能在程序 B 恢复以前已经向它自己的窗口输出新的内容，结果当程序 B 恢复它保存的窗口内容时，保存的内容可能是过时的（而 DOS 的情况就不同，TSR 程序激活的时候，用户程序是被挂起的），所以最好的办法就是让程序 A 自己来决定如何恢复。

Windows 系统采用的方法是：当 Windows 检测到窗口被覆盖的地方需要恢复的时候，它会向用户程序发送一个 WM_PAINT 消息，消息中包括了需要恢复的区域，然后由用户程序来决定如何恢复被覆盖的内容。

如果程序因为忙于处理其他事务以至于无法及时响应 WM_PAINT 消息，那么窗口客户区原先被覆盖的地方可能会被 Windows 暂时画成一块白色（或者背景色）的矩形，或者根本就是保留被覆盖时的情形，直到程序有时间去响应 WM_PAINT 消息为止。我们常常可以看到这种情况发生在死锁程序的客户区内，这就是因为死锁的程序无法响应 WM_PAINT 消息来恢复客户区造成的。

所以对于“*When*”这个问题，答案是：程序应该在 Windows 要求的时候绘画客户区，也就是在收到 *WM_PAINT* 消息的时候。如果程序需要主动刷新客户区，那么可以通过调用 *InvalidateRect* 等函数引发一条 *WM_PAINT* 消息，因为在 *WM_PAINT* 消息中刷新客户区的代码是必须存在的，所以用这种看似“舍近求远”的办法实际上可以节省一份重复的代码。即使是在游戏程序这种“主动刷新”远远多于“被动刷新”的程序中，只要窗口有被其他东西覆盖的可能，那么这个原则就是适用的。

2. GDI 程序的结构

对于 Win32 程序来说，*WM_PAINT* 消息随时可能发生，这就意味着，程序再也不能像在 DOS 下一样输出结果后就不管了，反过来，程序在任何时刻都应该知道如何恢复整个或局部客户区中以前输出的内容，本着这个要求，可以按图 7.1 所示来安排程序结构。

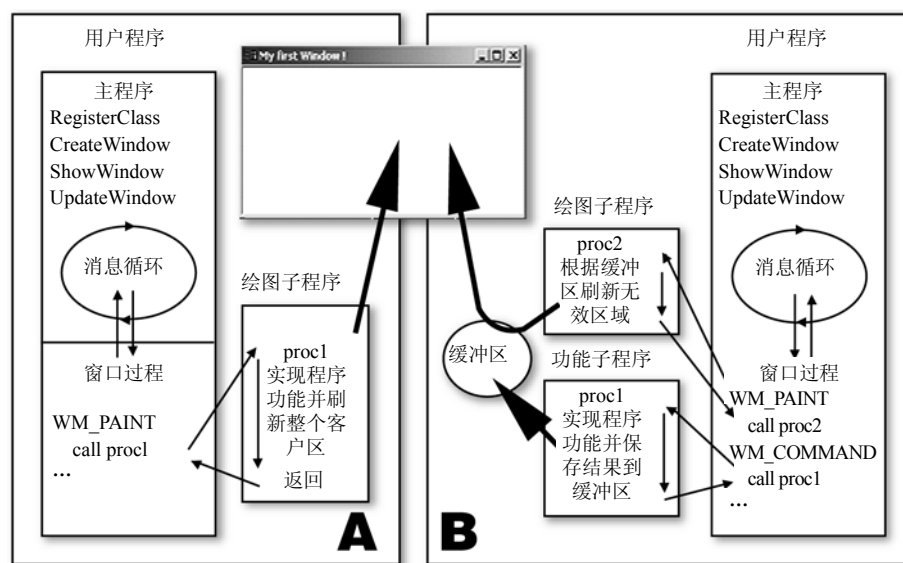


图 7.1 GDI 程序的结构

如果程序的功能比较简单，可以采取图中左边的 A 程序结构，即计算及刷新整个客户区的代码全部安排在 *WM_PAINT* 消息中完成，这样，每次当客户区的全部或部分需要被更新的时候，程序重新执行整个生成客户区屏幕数据的功能模块并刷新客户区。这种结构适用于功能模块很短小且执行速度很快的情况，整个过程的时间最好不超过几百 ms，否则，用户会在一个明显的等待时间后才看到程序把客户区中的“空洞”补上。考虑一个极端的情况：当程序输出的内容是经过千辛万苦才算出来的——这不是一件奇怪的事情，计算圆周率的程序就要动辄计算几个小时——那么即使客户区被别的窗口覆盖掉一点点，程序也要经过整个计算过程后才能重画客户区，而且在这个过程中，程序还没有从 *WM_PAINT* 消息返回，以至于无法处理其他消息，结果程序就会以客户区中有个空洞的难看姿势呆在屏幕上一动不动达几个小时！

当生成屏幕数据的功能模块有些复杂的时候，如刚才计算圆周率的例子，就应该考虑采用图中 B 程序所示的结构了。在这个程序中，功能模块和客户区刷新模块分别在不同的子程序中

实现，功能模块单独用一个子程序完成，这个子程序可以由用户通过选择菜单项在 WM_COMMAND 消息中执行，也可以新建另外一个线程来完成，总之，它最后把计算结果放到一个缓冲区中，而每当客户区需要刷新时，程序在 WM_PAINT 消息中调用客户区刷新子程序，这个子程序从计算好的缓冲区中取出数据并输出到客户区中，由于单纯的屏幕刷新过程是很快，所以用户根本来不及看到客户区中的空洞。

在本章后面的内容中有两个时钟的例子：Clock.exe 和 BmpClock.exe，前面一个例子采用的是 A 结构，后面一个例子采用的是 B 结构，读者在阅读的时候可以比较一下它们在结构上的不同。

3. 探讨 WM_PAINT 消息

当客户区被覆盖并重新显示的时候，Windows 并不是在所有的情况下都发送 WM_PAINT 消息，下面是几种不同的情况：

- 当鼠标光标移过窗口客户区，以及图标拖过客户区这两种情况，Windows 总是自己保存被覆盖的区域并恢复它，并不需要发送 WM_PAINT 消息通知用户程序。
- 当窗口客户区被自己的下拉式菜单覆盖，或者被自己弹出的对话框覆盖后，Windows 会尝试保存被覆盖的区域并在以后恢复它，如果因为某种原因无法保存并恢复的话，Windows 会发送一个 WM_PAINT 消息通知程序。
- 别的情况造成窗口的一部分从不可见变到可见，如程序从最小化的状态恢复、其他的窗口覆盖客户区后移开、用户改变了窗口的大小和用户按动滚动条等，在这些情况下，Windows 会向窗口发送 WM_PAINT 消息。
- 一些函数会引发 WM_PAINT 消息，如 UpdateWindow，InvalidateRect，以及 InvalidateRgn 函数等。

窗口过程收到 WM_PAINT 消息后，并不代表整个客户区都需要被刷新，有可能客户区被覆盖的区域只有一小块，这个区域就叫做“无效区域”，程序只需要更新这个区域。

与 WM_TIMER 消息类似，WM_PAINT 消息也是一个低级别的消息，虽然它不会像 WM_TIMER 消息一样被丢弃，但 Windows 总是在消息循环空的时候才把 WM_PAINT 放入其中，实际上，Windows 为每个窗口维护一个“绘图信息结构”，无效区域的坐标就在其中，每当消息循环空的时候，如果 Windows 发现存在一个无效区域，就会放入一个 WM_PAINT 消息。

无效区域的坐标并不附带在 WM_PAINT 消息的参数中，在程序中有其他方法可以获取，WM_PAINT 消息只是通知程序有个区域需要更新而已，所以 Windows 也不会同时将两条 WM_PAINT 消息放入消息循环，当 Windows 要放入一条 WM_PAINT 消息的时候，如果发现已经存在一个无效区域了，那么它只需要把新旧两个无效区域合并计算出一个新的无效区域就可以了，消息循环中还是只需要一条 WM_PAINT 消息。

由于存在“无效区域”这样一个机制，所以程序在 WM_PAINT 消息中对客户区刷新完毕后工作并没有结束，如果不使无效区域变得有效，Windows 会在下一轮消息循环中继续放入一个 WM_PAINT 消息。也正是因为 Windows 仅仅根据是否存在“无效区域”来决定是否发送 WM_PAINT

消息，而不是根据程序是否执行了刷新过程，所以程序也可以不去刷新客户区，而是简单地用一个 `ValidateRect` 函数直接让客户区变得有效，以此来“欺骗”Windows 已经没有无效区域了，当 Windows 检查“绘图信息结构”的时候发现没有了无效区域，也就不会继续发送 `WM_PAINT` 消息了。

`WM_PAINT` 消息的处理流程一般是：

```
.if      eax ==    WM_PAINT ;eax 为 uMsg
    invoke    BeginPaint,hWnd,addr stPS
    ;刷新客户区的代码
    invoke    EndPaint,hWnd,addr stPS
    xor      eax,eax
    ret
```

读者可以发现中间并没有调用 `ValidateRect` 来使无效区域变得有效，这是因为 `BeginPaint` 函数和 `EndPaint` 函数隐含有这个功能，如果不是以 `BeginPaint/EndPaint` 当做消息处理代码的头尾的话，那么在 `WM_PAINT` 消息返回的时候就必须调用 `ValidateRect` 函数。

`BeginPaint` 函数的第二个参数是一个绘图信息结构的缓冲区地址，Windows 会在这里返回绘图信息结构，结构中包含了无效区域的位置和大小，绘图信息结构的定义如下：

<code>PAINTSTRUCT</code>	<code>STRUCT</code>	
<code>hdc</code>	<code>DWORD</code>	<code>?</code>
<code>fErase</code>	<code>DWORD</code>	<code>?</code>
<code>rcPaint</code>	<code>RECT</code>	<code><></code>
<code>fRestore</code>	<code>DWORD</code>	<code>?</code>
<code>fIncUpdate</code>	<code>DWORD</code>	<code>?</code>
<code>rgbReserved</code>	<code>BYTE</code>	<code>32 dup(?)</code>
<code>PAINTSTRUCT</code>	<code>ENDS</code>	

其中 `hdc` 字段是窗口的设备环境句柄（在下一节中将要讲到），`rcPaint` 字段是一个 `RECT` 结构，它指定了无效区域矩形的对角顶点，`fErase` 字段如果为非零值，表示 Windows 在发送 `WM_PAINT` 消息前已经用背景色擦除了无效区域，后面 3 个字段是 Windows 内部使用的，应用程序不必去理会它们。

7.1.2 设备环境

好了，解决了“`When`”的问题，让我们来考虑一个新的问题，在 DOS 操作系统中，向屏幕输出数据实际上是把输出内容拷贝到视频缓冲区中，在第 1 章的图 1.1 中就已经说明：如果在文本模式下显示信息，只需要把内容拷贝到 `B8000h` 处的内存中；显示图形信息，可以把图形数据拷贝到 `A0000h` 处的内存中。

在 Windows 中，GDI 接口把程序和硬件分隔开来，在 Win32 编程中，再也不能通过直接向视频缓冲区拷贝数据的办法来显示信息了，那么，究竟该往哪里输出图形呢——这就是“`Where`”的问题。答案是：通过“设备环境”来输出图形。

1. 什么是设备环境

在实际使用中，通过“设备环境”可以操作的对象很广泛，除了可以是打印机或绘图仪等硬件设备外，也可以是窗口的客户区，包括大大小小的所有可以被称为窗口的按钮与控件等的客户区，也可以是一个位图。总之，任何需要用到图形操作的对象都可以通过“设备环境”进行绘图。

哭

```

_WinMain      proc
               local    @stWndClass:WNDCLASSEX
               local    @stMsg:MSG
               local    @hTimer

               invoke    GetModuleHandle, NULL
               mov       hInstance, eax
               invoke    RtlZeroMemory, addr @stWndClass, sizeof @stWndClass
;*****
               invoke    LoadCursor, 0, IDC_ARROW
               mov       @stWndClass.hCursor, eax
               push hInstance
               pop       @stWndClass.hInstance
               mov       @stWndClass.cbSize, sizeof WNDCLASSEX
               mov       @stWndClass.style, CS_HREDRAW or CS_VREDRAW
               mov       @stWndClass.lpfWndProc, offset _ProcWinMain
               mov       @stWndClass.hbrBackground, COLOR_WINDOW + 1
               mov       @stWndClass.lpszClassName, offset szClass1
               invoke    RegisterClassEx, addr @stWndClass
               invoke    CreateWindowEx, WS_EX_CLIENTEDGE, offset szClass1, \
               offset szCaption1, WS_OVERLAPPEDWINDOW, \
               450, 100, 300, 300, \
               NULL, NULL, hInstance, NULL
               mov       hWin1, eax
               invoke    ShowWindow, hWin1, SW_SHOWNORMAL
               invoke    UpdateWindow, hWin1
;*****
               mov       @stWndClass.lpszClassName, offset szClass2
               invoke    RegisterClassEx, addr @stWndClass
               invoke    CreateWindowEx, WS_EX_CLIENTEDGE, offset szClass2, \
               offset szCaption2, WS_OVERLAPPEDWINDOW, \
               100, 100, 300, 300, \
               NULL, NULL, hInstance, NULL
               mov       hWin2, eax
               invoke    ShowWindow, hWin2, SW_SHOWNORMAL
               invoke    UpdateWindow, hWin2
;*****
; 设置定时器
;*****
               invoke    SetTimer, NULL, NULL, 100, addr _ProcTimer
               mov       @hTimer, eax
;*****
; 消息循环
;*****
               .while    TRUE
               invoke    GetMessage, addr @stMsg, NULL, 0, 0
               .break    .if eax == 0
               invoke    TranslateMessage, addr @stMsg
               invoke    DispatchMessage, addr @stMsg
               .endw
;*****
; 清除定时器
;*****
               invoke    KillTimer, NULL, @hTimer

```

这个程序能演示出什么效果来呢？图 7.2 就是程序运行的结果，屏幕上的两个并排的正方形窗口就是 DcCopy 程序建立的窗口，程序每 100 ms 将右边窗口的客户区拷贝到左边的窗口客户区中，通过左边窗口的客户区就可以了解右边客户区 DC 对应的究竟是什么内容。



这个例子验证了“设备环境”只是“环境”而不是“设备”，它并不存储发给它的图形数据，图形数据透过它写到了它所描述的“设备”上，每个窗口客户区的“设备环境”对应的设备都是屏幕，但由于它们在位置上可能重叠，所以向一个窗口的客户区写数据相当于同时写了下层窗口的客户区。

为了让当前激活的窗口在视觉上保持在最上面，下层窗口向自己客户区写的内容首先要经过 Windows 的“过滤”，只有没有被其他窗口覆盖掉的部分才真正被写到了屏幕上。

读者应该时刻提醒自己——“设备环境”只是一个环境，是设备属性的一组定义，程序输出的图形数据透过“设备环境”被定向到了具体的设备上，“设备环境”本身并不存储这些数据（在这里也可以看出 Device Context 中 Context 一词的含义：设备环境的上面是应用程序，下面是具体设备，而它是用来“联系上下关系”用的）。



读者可能认为：屏幕上的窗口就像放在桌面上的一张张纸，虽然一张纸可能暂时被另一张遮住，但纸上写的内容还是存在的，移开另一张纸就可以再次露出来。但实际情况是：桌面更像一个用粉笔写的公告黑板，一个窗口相当于划了一块空间写告示，写另一个告示的时候要把老告示的内容擦去一部分以便写新的内容，擦去的内容也就不存在了，如果要恢复老告示，那么必须把擦去的部分重新写上去。

2. 获取设备环境句柄

要想对任何设备绘图，首先必须获取设备的“设备环境句柄”（hDC），几乎所有的 GDI 函数的操作目标都是 hDC，在程序中得到一个 hDC 有几种方法。

最常用的方法是在 WM_PAINT 消息中用 BeginPaint 函数得到 hDC，WM_PAINT 消息的代码结构一般是：

```
.if      eax == WM_PAINT ;eax 为 uMsg
    invoke BeginPaint, hWnd, addr stPS
    ;刷新客户区的代码
    invoke EndPaint, hWnd, addr stPS
    xor     eax, eax
    ret
```

BeginPaint 函数的返回值就是需要刷新区域的 hDC。要注意的是：BeginPaint 返回的 hDC 对应的尺寸仅是无效区域，无法用它绘画到这个区域以外的地方去。由于窗口过程每次接收 WM_PAINT 消息时的无效区域可能都是不同的，所以这个 hDC 的值仅在 WM_PAINT 消息中有效，程序不应该保存它并把它用在 WM_PAINT 消息以外的代码中。基于同样的道理，BeginPaint 和 EndPaint 函数只能用在 WM_PAINT 消息中，因为只有这时候才存在无效区域。

程序中常常有这种需求，就是在非 WM_PAINT 消息中主动绘画客户区，由于 BeginPaint 和 EndPaint 函数必须在 WM_PAINT 消息中使用，所以这时必须用另外的方法获取 hDC，可以使用以下的方法：

```
invoke     GetDC, hWnd          ;获取 hDC
;返回值是 hDC
```

;绘图代码	
invoke	ReleaseDC, hWnd, hDc ;释放 hDC

GetDC 函数返回的 hDC 对应窗口的整个客户区，当使用完毕的时候，hDC 必须用 ReleaseDC 函数释放。对于用 GetDC 获取的 hDC，Windows 建议使用的范围限于单条消息内，当程序在处理某条消息的时候需要绘画客户区时，可以用 GetDC 获取 hDC，但在消息返回前，必须用 ReleaseDC 将它释放掉，如果在下一条消息中需要继续用到 hDC，那么必须重新用 GetDC 函数获取。

上面两种方法获取的 hDC 都是窗口的 hDC，如果要操作的是其他的对象，如打印机、位图等，就不能使用 BeginPaint 或 GetDC 函数了。当绘图的对象是一个设备的时候，可以用 CreateDC 函数来建立一个 DC：

invoke	CreateDC, lpszDriver, lpszDevice, lpszOutput, lpInitData
--------	--

lpszDriver 指向设备名称，如显示设备的设备名是 DISPLAY，打印机的设备名一般为 WINSPOOL，下面这几句代码建立的 DC 对应整个屏幕：

szDriver	db	"DISPLAY", 0
	...	
	invoke	CreateDC, addr szDriver, NULL, NULL, NULL
	mov	hDC, eax

当绘图对象是位图的时候，同样需要一个与位图句柄相联系的 DC，这时可以用函数 CreateCompatibleDC 来创建一个显示表面仅存在于内存中的 DC：

invoke	CreateCompatibleDC, hDc
--------	-------------------------

参数中的 hDC 是用来参考的 DC 句柄，如果指定的参数是 NULL，那么建立的 DC 将和当前屏幕的设置兼容，为了用 CreateCompatibleDC 建立的 DC 绘制一个位图，还需要用 SelectObject 函数将 hDC 和位图句柄联系起来。在这之后，通过 hDC 进行的绘图操作会将像素数据更新到位图中。

用 CreateDC 和 CreateCompatibleDC 函数建立的 hDC 在使用结束以后，必须用 DeleteDC 函数删除，注意这里不能用 ReleaseDC，这个函数是和 GetDC 配合用的。

用 BeginPaint/EndPaint，以及 GetDC 获取的 hDC 的使用时间不能超出本条消息，与此相比，用 CreateDC，以及 CreateCompatibleDC 建立的 hDC 就没有这个限制，可以在任何时刻建立它并且一直使用到不再需要为止。

7.1.3 色彩和坐标

1. Windows 中的色彩

可以表示的颜色总数由颜色深度决定，也就是存储每个像素所用的位数，各种显示设备可以显示的颜色总数可能大不相同，如果设备支持的颜色深度太浅，就会影响到图像的质量，会让人看起来觉得很粗糙和不自然。

一种颜色可以分解成红、绿、蓝三原色，所以可以用红、绿、蓝 3 个分量的组合来表示各种颜色。

当设备支持的颜色深度少于等于 8 位时（如 8 位（256 色）、4 位（16 色）、2 位（4 色）或 1 位（2 色）），总体位数太少，不足以用来表达 3 个颜色分量，这时系统建立一个色彩表，像素数据用来做索引在色彩表中获取颜色值，所以低于 8 位的颜色称为索引色。

只有当颜色深度大于 8 位的时候，像素数据中才直接包含红、绿、蓝 3 个分量。当颜色深度为 16 位的时候，红、绿、蓝各用 5 位表示，剩下的 1 位用做属性位，实际可以表示的颜色数目为 $2^{15}=32\ 768$ 种，16 位深度的彩色又称为 16 位色、高彩色或增强色。当颜色深度为 24 位的时候，3 个分量各用 8 位表示，实际可以表示的颜色数目为 $2^{24}=16\ 777\ 216$ 种，24 位深度的彩色又称为 24 位色、16M 色或真彩色。对于人的双眼来说，超过 16 位的颜色就已经很难分辨了。

在 Win32 的编程中，统一使用 32 位的整数来表示一个深度为 24 位的颜色，在这 32 位中只使用低 24 位，每一种原色分量占用 8 位，其中 0~7 位为红色，8~15 位为绿色，16~23 位为蓝色。在程序中用到一种颜色常数的时候，可以如下使用：

```
mov    eax, 红色+绿色*100h+蓝色*10000h ;将颜色放入 eax 中
```

当显示设备无法表示 24 位色的时候，Windows 会自动用设备可以显示的最接近的颜色来代替它，当显示设备的颜色深度比较低的时候，可以通过函数 GetNearestColor 来得知一种颜色（dwColor）会被系统替换成哪种颜色：

```
invoke    GetNearestColor, hDC, dwColor ;返回真正使用的颜色值
```

但是当显示设备颜色深度太低的时候，经过 Windows 自动转换的图像可能会让人觉得很自然，所以在有些时候，程序员可能希望预先得知设备的颜色深度，然后根据具体情况显示不同的图形。

显示设备的颜色深度可以用以下函数获取：

```
invoke    GetDeviceCaps, hDC, PLANES
mov       ebx, dwPlanes
invoke    GetDeviceCaps, hDC, BITSPIXEL
mul       ebx
mov       dwColorDepth, eax
```

第一个函数调用返回 DC 的色彩平面数，第二个函数调用返回每个像素的色彩位数，颜色深度最后可以通过 dwPlanes 乘以 dwBitsPixel 得到。

2. Windows 中的坐标系

要用 GDI 函数绘图，就必须首先了解这些函数使用的坐标系，在默认的状态下，Windows 坐标系以左上角做坐标原点，以右方当做 X 坐标的正方向，以下方当做 Y 坐标的正方向。坐标的数值用一个有符号的 16 位数来表示，范围从 -32 768~32 767，坐标的单位为像素，如图 7.3 所示。这种坐标系定义方法的好处是：窗口中每一点的坐标不会因为窗口的大小改变而改变，

图 7.3 Windows 中的默认坐标系

可以设置的参数包括坐标原点、坐标的逻辑单位和坐标的正方向等，参数中的 iMapMode 为新的映射方式，其可以选择的取值如表 7.1 所示，Windows 默认使用的映射方法为 MM_TEXT。

映 射 方 法	原 点	逻 辑 单 位	X 正 方 向	Y 正 方 向
MM_TEXT (默认方式)	左上	像素	右	下
MM_HIENGLISH	左上	0.001 英寸	右	上
MM_LOENGLISH	左上	0.01 英寸	右	上
MM_HIMETRIC	左上	0.01 毫米	右	上
MM_LOMETRIC	左上	0.1 毫米	右	上
MM_TWIPS	左上	1/1440 英寸	右	上
MM_ISOTROPIC	可变	可变 (x=y)	可变	可变
MM_ANISOTROPIC	可变	可变 (x!=y)	可变	可变

最后两种映射方式 MM_ISOTROPIC 和 MM_ANISOTROPIC 提供了更灵活的选择，设置为这两种映射方式后，程序可以继续调用 SetViewportOrgEx, SetViewportExtEx 和 SetWindowExtEx 函数来自由设置坐标系的原点、逻辑单位和坐标的正方向等所有参数。在其他映射方式下的时候，不能使用这 3 个设置函数，这时任何对它们的调用都会被忽略。

图 7.4 时钟程序的运行结果

[illegible]

源文件 Clock.asm 如下:

204

哭

206

哭

```

        invoke  SelectObject, _hDC, eax
        invoke  DeleteObject, eax
        movzx   eax, @stTime.wHour
        .if     eax >= 12
            sub     eax, 12
        .endif
        mov     ecx, 360/12
        mul     ecx
        movzx   ecx, @stTime.wMinute
        shr     ecx, 1
        add     eax, ecx
        invoke  _DrawLine, _hDC, eax, 30
;*****
        invoke  GetStockObject, NULL_PEN
        invoke  SelectObject, _hDC, eax
        invoke  DeleteObject, eax
        popad
        ret

_ShowTime    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_ProcWinMain proc uses ebx edi esi hWnd, uMsg, wParam, lParam
    local     @stPS:PAINTSTRUCT

    mov       eax, uMsg
    .if       eax == WM_TIMER
        invoke  InvalidateRect, hWnd, NULL, TRUE
    .elseif   eax == WM_PAINT
        invoke  BeginPaint, hWnd, addr @stPS
        invoke  _ShowTime, hWnd, eax
        invoke  EndPaint, hWnd, addr @stPS
    .elseif   eax == WM_CREATE
        invoke  SetTimer, hWnd, ID_TIMER, 1000, NULL
;*****
    .elseif   eax == WM_CLOSE
        invoke  KillTimer, hWnd, ID_TIMER
        invoke  DestroyWindow, hWnd
        invoke  PostQuitMessage, NULL
;*****
    .else
        invoke  DefWindowProc, hWnd, uMsg, wParam, lParam
        ret
    .endif
;*****
    xor       eax, eax
    ret

_ProcWinMain endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_WinMain    proc
    local     @stWndClass:WNDCLASSEX
    local     @stMsg:MSG

    invoke    GetModuleHandle, NULL

```


程序首先用标准的方法建立了一个窗口，在窗口的初始化消息 WM_CREATE 中用 SetTimer 建立了一个周期为 1 秒的定时器，用来在窗口的客户区中绘画时钟。这个定时器在 WM_CLOSE 消息中用 KillTimer 函数撤销。在定时器消息中，程序用 InvalidateRect 函数让整个客户区

失效，相当于让 Windows 在消息循环中放入一条 WM_PAINT 消息，整个时钟的绘画在 WM_PAINT 消息中完成。

在 WM_PAINT 消息中程序用标准的方法调用 BeginPaint 函数获取窗口客户区的 hDC，以便在上面绘画时钟，在消息返回的时候用 EndPaint 函数释放 hDC，两个函数的中间，程序把 hDC 传给 _ShowTime 子程序，由这个子程序完成整个绘画工作。

在第 6 章中已经讲到：因为获取系统时间不能依赖于 WM_TIMER 消息的计数，所以在 _ShowTime 子程序的开始，程序调用 GetLocalTime 来获取当前的系统时间，并根据这个时间来绘画时钟的时、分、秒指针。由于绘画的过程很快，所以整个程序的结构使用前面图 7.1 中所示的 A 结构，也就是每次有 WM_PAINT 消息的时候，程序总是重画整个客户区，所以读者在速度比较慢的计算机上运行这个程序时，可能会看到有个闪烁的过程，因为程序每次总是先将整个客户区清除成背景色（InvalidateRect 函数最后的 TRUE 参数要求 Windows 在发送 WM_PAINT 消息前清除客户区），然后绘画四周的刻度，最后画上指针。绘画刻度是由 _DrawDot 子程序完成的，绘画指针是由 _DrawLine 子程序完成的。

GetLocalTime 后面的 _CalcClockParam 子程序根据客户区的尺寸计算时钟尺寸参数，它比较客户区高度和宽度，以其中的较小值用做时钟的直径，计算得到的圆心最后存放于全局变量 dwCenterX 和 dwCenterY 中，计算得到的半径存放于 dwRadius 中。

程序中有两个公用的子程序：_CalcX 和 _CalcY，它们用来计算角度对应的坐标，如图 7.5 所示，时钟 0 点时间是从垂直方向开始的，以时间值为角度配合 Windows 的默认坐标系，对应某个时间点 (x, y)，x 应该是圆心 x 加上角度的正弦值乘以半径，y 应该是圆心 y 减去角度的余弦值乘以半径。_CalcX 和 _CalcY 输入的参数是角度 _dwDegree 和半径 _dwRadius。子程序中使用 80×86 的协处理器指令，首先将角度值换算成弧度值——乘以 π 并除以 180，然后用上面分析的公式进行浮点计算并将结果返回。

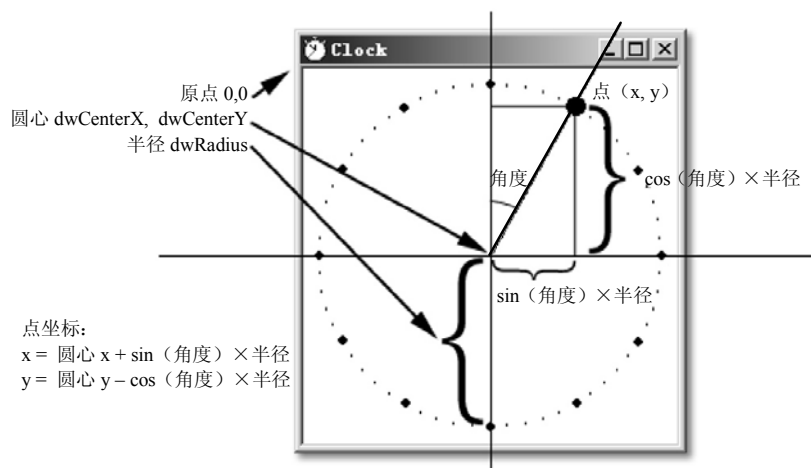


图 7.5 时钟程序的坐标计算

在接下来的内容中，先介绍一些绘画操作的背景知识。

7.2.1 画笔和画刷

GDI 中的绘画函数有 3 大类：画点、画线和画填充区域。使用过 Photoshop 等图形软件的读者一定知道，在画线之前需要选择一种画笔，这样画出来的线条都是基于这种画笔的；同样，填充一个区域之前需要选择一种画刷，这样整个填充区域将重复使用这个画刷的颜色或图案。

GDI 中也有同样的画笔和画刷的概念，画笔、画刷，以及其他一些 GDI 中要使用的东西，包括字体、区域、路径、图案和位图统称 GDI 中的“对象”，通过 SelectObject 函数可以指定一个 DC 当前使用的对象对应哪个对象句柄，称为“当前对象”，当设置了一个当前对象的时候，以后和这种对象相关的函数都将使用当前对象，直到再次用 SelectObject 选择新的对象为止。比如，当选择了新的画笔后，以后所有画线函数画出来的线条样式都是由这个画笔决定的，而选择了新的画刷后，则所有填充函数填充的样式都将使用这个画刷。

SelectObject 函数的用法是：

invoke	SelectObject, hDC, hGDIOBJECT
mov	hOldObject, eax

其中参数 hGDIOBJECT 就是对象的句柄，它可以是位图句柄、画笔句柄、画刷句柄、字体句柄或区域句柄，函数会根据句柄的种类自动替换原有的对象，并将原来使用的对象句柄返回（当对象类型是区域的时候除外），如果 DC 中原来没有设置当前对象，那么函数的返回值是 GDI_ERROR 或 NULL。

1. 使用预定义的画笔和画刷

Windows 预定义了一些常用的画笔和画刷，在程序中可以用 GetStockObject 来获取它们的句柄，Stock 的中文含义是“常备的、库存的”，所以这个函数字面上的意思就是“获取常用的对象”，注意并没有类似于 GetStockPen 或 GetStockBrush 之类的函数，所有获取常用对象的操作统一使用 GetStockObject 函数。

GetStockObject 函数的用法是：

invoke	GetStockObject, fnObject
mov	hObject, eax

fnObject 参数是预定义的对象类型，可以是表 7.2 所示的取值。

表 7.2 GDI 中的常用对象

预 定 义 值	说 明
BLACK_PEN	黑色画笔
WHITE_PEN	白色画笔
NULL_PEN	空画笔
BLACK_BRUSH	黑色画刷
续表	
预 定 义 值	说 明

DKGRAY_BRUSH	深灰色画刷
GRAY_BRUSH	灰色画刷
LTGRAY_BRUSH	浅灰色画刷
WHITE_BRUSH	白色画刷
HOLLOW_BRUSH 或 NULL_BRUSH	空画刷
ANSI_FIXED_FONT	等宽系统字体
ANSI_VAR_FONT	不等宽系统字体
DEFAULT_GUI_FONT (Win95)	默认系统字体（用于菜单、对话框等）
OEM_FIXED_FONT	OEM 等宽字体
SYSTEM_FONT	默认系统字体（用于菜单、对话框等）
DEFAULT_PALETTE	默认图案

NULL_PEN 和 NULL_BRUSH 是空画笔和空画刷，之所以有空的对象，是因为绘制填充区域的函数同时用到了画笔和画刷——绘制的外框使用当前画笔，中间用当前画刷填充。使用空对象可以有机会画出没有边框线只有填充图案，或者只有边框线而不填充的区域来。

用 GetStockObject 函数得到对象句柄以后，就可以用 SelectObject 函数将对象句柄设置到 DC 中了。例子文件 Clock.asm 中的 _ShowTime 函数中用 GetStockObject 函数获取了一个 BLACK_BRUSH 画刷，用来绘画时钟的刻度。

2. 使用自定义的画笔和画刷

使用 GetStockObject 函数得到的对象是最“简陋”的，如画笔只能是白色或黑色的宽度为 1 像素的实线，画刷只能是白色、黑色和有限的几种灰色色块。要想使用彩色的、多种多样风格的画笔和画刷，就必须用自定义的方法。

创建自定义的画笔可以使用 CreatePen, ExtCreatePen 或 CreatePenIndirect 函数，CreatePen 函数的使用方法是：

invoke	CreatePen, fnPenStyle, dwWidth, dwColor
mov	hPen, eax

fnPenStyle 参数是画笔风格，它可以是两种实线风格 PS_SOLID, PS_INSIDEFRAME 或空画笔 PS_NULL，以及几种虚线风格 PS_DASH, PS_DOT, PS_DASHDOT 或 PS_DASHDOTDOT。它们对应的线条如图 7.6 所示，图中从上到下分别是 PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT 和 PS_INSIDEFRAME 风格的线条，几种虚线的风格很好记，只要记得“点”就是 DOT，“划”就是 DASH 就可以了，如 PS_DASHDOTDOT 风格就是由“划、点、点”重复组成的虚线。

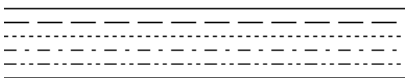


图 7.6 几种自定义画笔风格

PS_SOLID 和 PS_INSIDEFRAME 风格的画笔使用的都是实线线条，它们之间的区别在于当画笔的宽度大于 1 像素。在使用区域绘画函数的时候，PS_SOLID 线条会居中画于边线上，而 PS_INSIDEFRAME 线条会全部画在边线里面，它的宽度会向区域的内部扩展，所以它的名称是 INSIDEFRAME。

CreatePen 函数的 dwWidth 参数定义了画笔的宽度，单位是 DC 坐标映射方法中定义的逻辑单位，如果这个参数使用 NULL，那么函数会使用 1 像素的宽度。宽度参数会影响到风格参数：

当宽度大于 1 的时候，画笔风格不能使用虚线，这时候即使指定了虚线风格，函数也会自动使用 PS_SOLID 风格。dwColor 参数指定了画笔的颜色。

例子源代码的_ShowTime 子程序中用不同宽度的线条来绘画时、分、秒指针，绘画前就使用 CreatePen 函数创建了不同宽度的画笔。

如果需要创建更复杂的画笔，可以使用 ExtCreatePen 函数。这个函数除了有 CreatePen 的全部功能外，还可以让用户自己定义线条的样子，这样可以不必限制于上面的点点划划了。函数的用法读者可以参考函数手册。

创建自定义画刷可以使用的函数有：CreateSolidBrush, CreateHatchBrush, CreatePatternBrush 和 CreateBrushIndirect。

CreateSolidBrush 创建单色的画刷：

invoke	CreateSolidBrush, dwColor
mov	hBrush, eax

要输入的惟一参数是画刷的颜色。而 CreateHatchBrush 可以创建几种预定义图案的画刷：

invoke	CreateHatchBrush, iHatchStyle, dwColor
mov	hBrush, eax

dwColor 指定了图案线条的颜色，iHatchStyle 定义了不同的图案线条，这些图案线条实际上是以 8×8 的位图重复铺开组成的，iHatchStyle 的定义值可以是 HS_BDIAGONAL, HS_CROSS, HS_DIAGCROSS, HS_FDIAGONAL, HS_HORIZONTAL 和 HS_VERTICAL，这 6 种图案的花样在图 7.7 中从左到右排列显示。

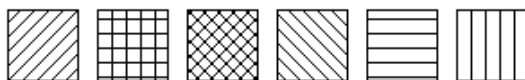


图 7.7 CreateHatchBrush 中的画刷图案

如果这些简单的图案不能满足使用要求，CreatePatternBrush 是个很好的选择：

invoke	CreatePatternBrush, hBitmap
mov	hBrush, eax

这个函数用一个位图当做画刷的图案，当要绘画的区域大于位图尺寸的时候，位图被重复铺开，就像 HTML 文件中的背景图案一样。读者可以尝试一下用一幅做网页文件背景的位图创建一个位图画刷，并且在 RegisterClassEx 时在 WNDCLASSEX 结构中的 hbrBackground 字段中使用这个画刷，这样创建出来的窗口背景会和网页背景一样华丽！演示代码请参考所附光盘的 Chapter07\TestObject 目录中的源代码。

对于自定义的画笔和画刷，还有其他自定义的对象，在不再需要的时候必须使用 DeleteObject 函数删除，但是要注意：当对象还是一个 DC 的当前对象的时候不要将它删除，在删除前应该确定 DC 中已经选入了其他的对象。与之相反，用 GetStockObject 获取的预定义对象使用后不需要删除，但是对它们调用 DeleteObject 也没有关系，因为它们不会被真正删

除。由于 `SelectObject` 返回值就是 DC 原来使用的对象句柄，所以删除对象的一个好时机就是当 `SelectObject` 返回的时候，如例子程序的 `_ShowTime` 子程序中用的：

invoke	CreatePen, PS_SOLID, 2, 0
invoke	SelectObject, _hDC, eax
invoke	DeleteObject, eax

`SelectObject` 将 `CreatePen` 创建的画笔句柄选入 DC，返回值 `eax` 就是以前使用的画笔句柄，这个句柄不再使用了，所以可以在下面用 `DeleteObject` 直接删除，而这次建立的画笔可以在下次执行 `SelectObject` 后用同样的方法删除。

7.2.2 绘制像素点

在 DC 上绘制像素点是绘图最基本的操作，使用的方法是：

invoke	SetPixel, hDC, dwX, dwY, dwColor
--------	----------------------------------

`SetPixel` 函数在 hDC 的 `dwX`、`dwY` 位置以 `dwColor` 为颜色画上一个像素点，如果需要获取 hDC 中某个像素点当前的颜色值，那么可以使用 `GetPixel` 函数：

invoke	GetPixel, hDC, dwX, dwY
mov	dwColor, eax

虽然绘画像素是最基本的绘图操作方法，但是在程序中一般很少使用 `SetPixel` 函数，因为它的开销太大了，只适合用在需要少量绘画像素的地方，如果要绘画一个线条或者整个区域，那么最好使用画线函数或者填充函数，因为这些函数是在驱动程序级别上完成的，所有的硬件加速功能都可以用上。

图形处理前最基本的步骤是获取像素，但也不应该用 `GetPixel` 函数来获取一大块的像素数据，理由是同样的。如果要分析整个区域的像素数据，最好的办法就是用 `GetDIBits` 函数将全部数据拷贝到内存中再进行处理。

7.2.3 绘制图形

GDI 的图形绘制函数主要有绘制线条和填充区域两大类。绘制线条的函数以当前画笔绘制线条；绘制填充区域的函数以当前画笔绘制边线，并以当前画刷填充中间的区域。

1. 绘制线条

绘制线条的函数有画直线的 `LineTo`，画多条直线的 `Polyline` 和 `PolylineTo`，画贝塞尔曲线的 `PolyBezier` 和 `PolyBezierTo`，画弧线的 `Arc` 和 `ArcTo`。

DC 的数据结构中有一个“当前点”，`LineTo` 函数就是从当前点画一条直线到参数中指定的点，并把参数中指定的点设置为新的当前点。画线函数中所有以 `To` 结尾的函数都是从当前点开始绘制的，如 `LineTo`，`PolylineTo`，`PolyBezierTo` 和 `ArcTo`，由于这些函数在绘画结束后会把绘制的最后一点设置为新的当前点，所以在使用这些函数的时候要考虑到当前点也是参与绘制的坐标一部分。而其余的 `Polyline`，`PolyBezier` 和 `Arc` 函数则和当前点没有关系，也不会影响当前点的位置。

如果要设置当前点的位置，可以使用 MoveToEx 函数：

```
invoke    MoveToEx, hDC, dwX, dwY, lpPoint
```

dwX 和 dwY 指出了新的当前点的坐标，lpPoint 指向一个空的 POINT 结构，用来返回原来的当前点位置，如果不需要的话，这个参数可以使用 NULL。

另一个函数也可以得到当前点的坐标：

```
invoke    GetCurrentPositionEx, hDC, lpPoint
```

同样，lpPoint 指向一个用来返回当前点坐标的 POINT 结构地址。

如果要绘制一条直线，必须配合使用 MoveToEx 和 LineTo 函数，首先由 MoveToEx 函数设置一个当前点当做起始坐标，然后用 LineTo 绘画到结束坐标，如 Clock.asm 中的 DrawLine 子程序中就是这样绘制时钟指针的：

```
invoke    MoveToEx, _hDC, @dwX1, @dwY1, NULL
invoke    LineTo, _hDC, @dwX2, @dwY2
```

这两句代码绘画一条从 @dwX1, @dwY1 到 @dwX2, @dwY2 的直线。

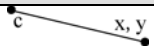
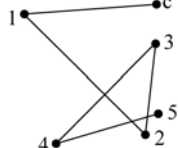
如果要绘制是相连的多条直线，可以使用 Polyline 或 PolylineTo 函数：

```
invoke    PolylineTo, hDC, lpPoint, cPoints
invoke    Polyline, hDC, lpPoint, cPoints
```

lpPoint 指向一个包含一系列 POINT 结构的缓冲区，由于 POINT 结构只有 X 和 Y 两个字段，所以缓冲区中的数据实际上是 x1, y1, x2, y2, x3, y3, …，cPoints 参数指出了点的数目，注意：PolylineTo 画出的直线是从当前点坐标 (x, y) 开始，然后到 (x1, y1)，再到 (x2, y2)，…，而 Polyline 函数画出的直线是从 (x1, y1) 开始的，对于这个函数，如果 cPoints 参数指定了 n 个点，那么直线的数量实际上是 n-1。当绘制的相连直线很多的时候，用 Polyline 或 PolylineTo 比多次使用 LineTo 的速度要快很多，就像用填充函数比多次使用 SetPixel 要快一样。

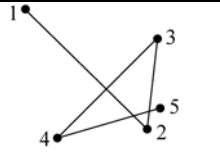
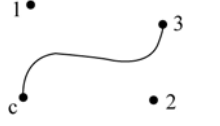
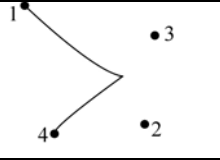
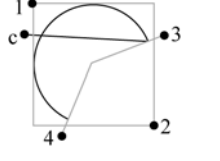
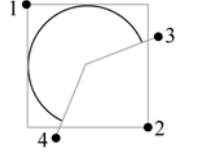
表 7.3 举例说明了这些画线函数的功能，表中的 (x1, y1) 或 (x2, y2) 等表示点 1 或点 2 的坐标，(xc, yc) 表示当前点的坐标，当前点在图中用 c 表示。

表 7.3 画线函数的功能

函 数	说 明	图 例
LineTo(hDC, x, y)	从当前点到 (x, y) 点	
PolylineTo(hDC, lpPoint, 5)	lpPoint 指向存放 (x1, y1) 到 (x5, y5) 的缓存区，函数画的线条从 (xc, yc) 到 (x1, y1) 到 (x2, y2) …到 (x5, y5)，共 5 条直线	

续表

函 数	说 明	图 例
-----	-----	-----

<code>Polyline(hDC, lpPoint, 5)</code>	lpPoint 指向存放 (x1, y1) 到 x5、y5 的缓存区，函数画的线条从 (x1, y1) 到 (x2, y2) …到 (x5, y5)，共 4 条直线	
<code>PolyBezierTo(hDC, lpPoint, 3)</code>	绘画的 Bezier 曲线的控制点为 (xc, yc) 和 (x1, y1) 和 (x2, y2) 和 (x3, y3)	
<code>PolyBezier(hDC, lpPoint, 4)</code>	绘画的 Bezier 曲线的控制点为 (x1, y1) 和 (x2, y2) 和 (x3, y3) 和 (x4, y4)	
<code>ArcTo(hDC, x1, y1, x2, y2, x3, y3, x4, y4)</code>	首先画 (xc, yc) 到起始点的直线，再画起始点到结束点的弧线	
<code>Arc(hDC, x1, y1, x2, y2, x3, y3, x4, y4)</code>	画起始点到结束点的弧线	

对于 Arc 和 ArcTo 函数，参数 (x1, y1) 和 (x2, y2) 定义了一个矩形的对角点，然后在和这个矩形相切的椭圆上面，以椭圆的中心（也就是矩形的中心）画两条假想的直线到 (x3, y3) 和 (x4, y4)，这两条直线和椭圆相交的点就是圆弧的起始点和结束点。在默认情况下，圆弧由起始点沿着椭圆从逆时针方向画到结束点。不过绘画方向可以由 SetArcDirection 函数重新规定：

invoke	SetArcDirection, hDC, AD_COUNTERCLOCKWISE	;逆时针方向
invoke	SetArcDirection, hDC, AD_CLOCKWISE	;顺时针方向

读者一定注意到了一个问题：在画线的时候，如果当前的画笔是虚线的话，虚线的不连续部分实际上是由白色组成的，当虚线画在非白色的背景上的时候这一点显得特别明显。实际上，可以选择这些不连续部分的颜色，用以下的语句就可以做到这一点：

invoke	SetBkColor, hDC, dwColor	
--------	--------------------------	--

调用后不连续的部分就将用 dwColor 指定的颜色绘画。

但是改变颜色也并不是惟一的选择，GDI 允许这部分并不绘画任何颜色，也就是可以是“透明”的，用下面的调用可以将模式在透明和非透明之间切换：

invoke	SetBkMode, hDC, OPAQUE	;非透明模式
invoke	SetBkMode, hDC, TRANSPARENT	;透明模式

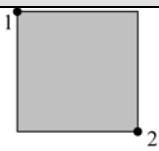
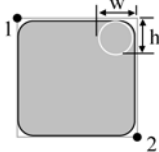
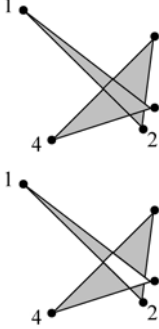
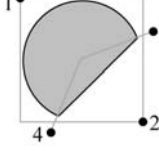

两种模式以及绘画颜色不单影响虚线的空隙部分，同样也影响 CreateHatchBrush 函数创建的画刷，因为这种画刷使用几种由线条构成的图案，当用这种画刷填充一个区域的时候，线条图案的空隙部分同样受 SetBkColor 函数和 SetBkMode 函数的影响。

2. 绘制边界框和填充区域

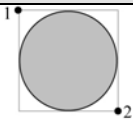
绘制边界框和填充区域其实是同一件事情。如果当前画笔是 NULL_PEN 的话，画出来的是没有边线的填充区域；如果当前画刷是 NULL_BRUSH 的话，那么只有边线而不会填充；如果当前画刷既不是 NULL_PEN 也不是 NULL_BRUSH，那么画出来的图形既有边线也是填充的。

绘制区域的函数有画矩形的 Rectangle，画圆角矩形的 RoundRect，画多边形的 Polygon，画弦的 Chord，画圆饼的 Pie 和画椭圆的 Ellipse。这些函数的使用效果如图 7.4 所示。

表 7.4 填充函数的功能

函 数	说 明	图 例
Rectangle(hDC, x1, y1, x2, y2)	画以 (x1, y1) 和 (x2, y2) 为对角坐标的填充矩形	
RoundRect(hDC, x1, y1, x2, y2, w, h)	画以 (x1, y1) 和 (x2, y2) 为对角坐标的填充矩形，四个角以一个小椭圆来画圆角，小椭圆的宽和高为 w 和 h	
Polygon(hDC, lpPoint, 5)	lpPoint 指向存放 (x1, y1) 到 (x5, y5) 的缓存区，函数从 (x1, y1) 到 (x2, y2) ... 到 (x5, y5)，再回到 (x1, y1)，一共画 5 条直线并填充	
Chord(hDC, x1, y1, x2, y2, x3, y3, x4, y4)	以和 Arc 函数同样的方法画弧，然后连接弧的两个端点并填充	
Pie(hDC, x1, y1, x2, y2, x3, y3, x4, y4)	以和 Arc 函数同样的方法画弧，然后将弧的两个端点分别和椭圆中心连接并填充	

续表

函 数	说 明	图 例
Ellipse(hDC, x1, y1, x2, y2)	以 (x1, y1) 和 (x2, y2) 为对角定义一个矩形，然后画矩形相切的椭圆并填充	

在这些函数中，Polygon 的调用方式和 Polyline 很相似，只不过如果最后一点和第一点不同的话，函数自动再画一条和起始点相连的直线将整个区域闭合起来。用 Polygon 绘画的多边形中各条直线可能相交，Windows 允许程序自行选择填充的模式，可以是表 7.4 中 Polygon 一栏中的上面那个图例（填充全部区域），也可以是下面那个图例（间隔填充区域）。可以用下面的函数切换填充的模式：

invoke	SetPolyFillMode, _hDC, ALTERNATE ;间隔填充
invoke	SetPolyFillMode, _hDC, WINDING ;填充全部区域

Chord 函数和 Pie 函数的参数使用和画弧线的 Arc 函数相似，只不过 Chord 函数将弧线的两端直接相连，形成一个“弦”，而 Pie 函数将两端和圆心相连，形成一个“圆饼”，这两个函数绘画的方向同样受 SetArcDirection 函数设置的影响。

在例子 Clock.asm 中，程序在 DrawDot 子程序中用 Ellipse 函数绘画时钟的刻度，读者也可以将程序改动一下，尝试着用 Polygon 画五角星来当做时钟的刻度。

除了这些函数，还有 3 个和矩形有关的填充函数：FillRect，FrameRect 和 InvertRect，这些函数不使用当前画笔画边线，也不用当前画刷填充，其中 FillRect 函数用指定的画刷 hBrush 填充一个 lpRect 指定的矩形区域，lpRect 指向一个 RECT 结构；FrameRect 函数用指定画刷 hBrush 绘画边线；InvertRect 函数将 lpRect 指定的矩形区域中的颜色值取反。用法如下：

invoke	FillRect, hDC, lpRect, hBrush
invoke	FrameRect, hDC, lpRect, hBrush
invoke	InvertRect, hDC, lpRect

假设背景为白色，而参数中 hBrush 指定的画刷为灰色画刷，那么上述 3 个函数的运行结果如图 7.8 所示。



图 7.8 FillRect，FrameRect 和 InvertRect 函数的运行结果

图中左边是 FillRect 的运行结果，可以看到图案没有边线；中间是 FrameRect 的运行结果，它用灰色画刷绘画边线，得到了一个灰色的矩形边框；右边是 InvertRect 的运行结果，由于底色是白色的，白色取反得到的是黑色，所以整个矩形都变成了黑色。

7.2.4 绘图模式

在前面的内容中我们都是尝试在 DC 上用绘图函数画出需要的图形，对于 DC 上被绘画上去的像素来说，相当于用画笔（或画刷）的像素点代替了原来的像素点，但 Windows 也可以用画

笔的像素点和原来的像素点进行计算以后的值当做新的像素点，这个计算的过程就叫做光栅运算，光栅运算的方法用“光栅运算符”来定义——英文缩写是 ROP（Raster Operation），ROP 码是一些取反、异或、拷贝、或及与等位运算方法的组合。

对于绘图函数，Windows 定义了 16 种 ROP 码，如表 7.5 所示。表中的“像素”指 DC 中要绘画位置原来的像素值，画笔指要画上去的颜色值，当然 ROP 码影响的并不单是画笔画出的线条，同样影响用画刷填充的区域，所以读者不要被表中的“PEN”搞混淆了，这个“PEN”指的是“Pen and Brush”！

ROP 为一些应用提供了方便，比如需要在背景上拖动一个图形，如果用普通的绘画方法，那么在绘画前必须保存原来背景的数据，在图形拖动后再恢复，然后在新的位置再保存、再绘画，如此重复。但如果使用 R2_XORPEN 或 R2_NOTXORPEN 的绘画模式，因为 xor 操作两遍就是原来的数值，所以无须保存原来的像素，在相同的地方再绘画一遍就相当于恢复原来的图形。而用 R2_BLACK 和 R2_WHITE 就相当于不管画笔和画刷是什么颜色，画出来的全部是黑色或白色。

表 7.5 绘图模式中可以使用的 ROP 码

ROP 码	新像素点算法	说 明
R2_BLACK	0	总为黑色
R2_WHITE	1	总为白色
R2_NOP	像素	保持不变
R2_NOT	not(像素)	原来像素的颜色取反
R2_COPYPEN	画笔	画笔颜色
R2_NOTCOPYPEN	not(画笔)	画笔颜色取反
R2_MERGEENNOT	画笔 or not(像素)	画笔颜色与原像素颜色取反后值的复合
R2_MASKPENNOT	画笔 and not(像素)	画笔和原像素取反后值的共同色
R2_MERGEENNOTPEN	像素 or not(画笔)	原像素颜色与画笔取反颜色的复合
R2_MASKNOTPEN	像素 and not(画笔)	原来像素和画笔取反后的共同色
R2_MERGEEN	像素 or 画笔	画笔颜色与原来像素的复合
R2_NOTMERGEEN	not(像素 or 画笔)	R2_MERGEEN 再取反
R2_MASKPEN	像素 and 画笔	画笔和原来像素的共同色
R2_NOTMASKPEN	not(像素 and 画笔)	R2_MASKPEN 再取反
R2_XORPEN	像素 xor 画笔	画笔和原来像素的异或值
R2_NOTXORPEN	not(像素 xor 画笔)	R2_XORPEN 再取反

对于一个 DC 来说，默认的绘图模式是 R2_COPYPEN，就是用画笔或画刷的颜色替换掉原来像素的颜色。如果要设置新的绘图模式，可以使用 SetROP2 函数。如下面的语句将绘图模式设置为 R2_NOTCOPYPEN 模式，这样以后的所有的绘图函数就将以画笔或画刷取反后的颜色绘图了：

```
invoke SetROP2, hDC, R2_NOTCOPYPEN
```

如果要获取当前的绘图模式，可以使用 GetROP2 函数，函数返回当前的模式：

```
invoke GetROP2, hDC
```

7.3 创建和使用位图

7.2 节探讨了绘制图形的一些函数，虽然绘图操作是图形程序必不可少的一部分，但丰富多彩的界面大部分还是靠设计精美的位图来铺成的，而不是靠绘图函数一点点画出来的。在大部分程序中，使用预先设计好的位图是最普遍的做法，在这一节中，将讨论如何使用位图，并在下一节中讨论使用块传送函数对位图进行操作。

7.3.1 一个使用位图的时钟例子

本节使用另一个时钟的例子，这个时钟的背景和边框用位图组成，程序中有两套背景图片和两套边框图片可供自由选择，图 7.9 显示了几种不同组合下的时钟外形，最右边的是在时钟上面按下右键弹出的选择菜单。

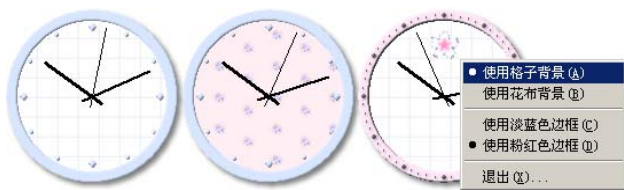


图 7.9 BmpClock 时钟程序的运行结果

程序的源代码可以在所附光盘的 Chapter07\BmpClock 目录中找到，包括汇编源程序 BmpClock.asm、资源脚本文件 BmpClock.rc 和一些图片。

BmpClock.rc 源文件如下，中间定义了一些程序中要使用的位图：

```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#include      <resource.h>
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

#define       ICO_MAIN          100
#define       IDC_MAIN          100
#define       IDC_MOVE          101
#define       IDB_BACK1         100
#define       IDB_CIRCLE1       101
#define       IDB_MASK1         102
#define       IDB_BACK2         103
#define       IDB_CIRCLE2       104
#define       IDB_MASK2         105
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

ICO_MAIN            ICON           "Main.ico"
IDC_MAIN            CURSOR        "Main.cur"
IDC_MOVE            CURSOR        "Move.cur"
IDB_BACK1           BITMAP        "Back1.bmp"
IDB_CIRCLE1         BITMAP        "Circle1.bmp"
IDB_MASK1           BITMAP        "Mask1.bmp"
IDB_BACK2           BITMAP        "Back2.bmp"
IDB_CIRCLE2         BITMAP        "Circle2.bmp"
IDB_MASK2           BITMAP        "Mask2.bmp"
```

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include                windows.inc
include                user32.inc
includelib             user32.lib
include                kernel32.inc
includelib             kernel32.lib
include                Gdi32.inc
includelib             Gdi32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Equ 等值定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
CLOCK_SIZE            equ        150
ICO_MAIN              equ        100
IDC_MAIN              equ        100
IDC_MOVE              equ        101
IDB_BACK1             equ        100
IDB_CIRCLE1           equ        101
IDB_MASK1             equ        102
IDB_BACK2             equ        103
IDB_CIRCLE2           equ        104
IDB_MASK2             equ        105
ID_TIMER              equ        1
IDM_BACK1             equ        100
IDM_BACK2             equ        101
IDM_CIRCLE1           equ        102
IDM_CIRCLE2           equ        103
IDM_EXIT              equ        104
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .data?
hInstance             dd        ?
hWinMain              dd        ?
hCursorMove           dd        ?           ;Cursor when move
hCursorMain           dd        ?           ;Cursor when normal
hMenu                 dd        ?
hBmpBack              dd        ?
hDcBack               dd        ?
hBmpClock             dd        ?
hDcClock              dd        ?
dwNowBack             dd        ?
dwNowCircle           dd        ?
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .const
szClassName           db        'Clock', 0
dwPara180             dw        180
dwRadius              dw        CLOCK_SIZE/2

```

222

哭

224

哭

226

```

        mov     eax,hWnd
        mov     hWinMain,eax
        invoke  _Init
;*****
        .elseif eax == WM_COMMAND
            mov     eax,wParam
;*****
; 由于印刷宽度的问题，影响源代码的缩进格式，请读者注意
;*****
        .if     ax == IDM_BACK1
            mov     dwNowBack, IDB_BACK1
            invoke  CheckMenuItem, hMenu, \
                IDM_BACK1, IDB_BACK1, NULL
        .elseif ax == IDM_BACK2
            mov     dwNowBack, IDB_BACK2
            invoke  CheckMenuItem, hMenu, \
                IDM_BACK1, IDB_BACK2, NULL
        .elseif ax == IDM_CIRCLE1
            mov     dwNowCircle, IDB_CIRCLE1
            invoke  CheckMenuItem, hMenu, IDM_CIRCLE1, IDB_CIRCLE2, \
                IDM_CIRCLE1, NULL
        .elseif ax == IDM_CIRCLE2
            mov     dwNowCircle, IDB_CIRCLE2
            invoke  CheckMenuItem, hMenu, IDM_CIRCLE1, IDB_CIRCLE2, \
                IDM_CIRCLE2, NULL
        .elseif ax == IDM_EXIT
            call _Quit
            xor     eax,eax
            ret
        .endif
;*****
; 恢复源代码缩进格式
;*****
            invoke  _DeleteBackGround
            invoke  _CreateBackGround
            invoke  _CreateClockPic
            invoke  InvalidateRect, hWnd, NULL, FALSE
        .elseif eax == WM_CLOSE
            call _Quit
;*****
; 按下右键时弹出一个 POPUP 菜单
;*****
        .elseif eax == WM_RBUTTONDOWN
            invoke  GetCursorPos, addr @stPos
            invoke  TrackPopupMenu, hMenu, TPM_LEFTALIGN, \
                @stPos.x, @stPos.y, NULL, hWnd, NULL
;*****
; 由于没有标题栏，下面代码用于按下左键时移动窗口
; UpdateWindow: 即时刷新，否则要等到放开鼠标时窗口才会重画
;*****
        .elseif eax == WM_LBUTTONDOWN
            invoke  SetCursor, hCursorMove
            invoke  UpdateWindow, hWnd
            invoke  ReleaseCapture

```

228

由于程序中的坐标算法与 7.2 节的 Clock.asm 是一样的, 所以有些子程序沿用了上一个程序中的相关子程序, 如计算坐标的 CalcX 和 CalcY, 绘画指针的 DrawLine 子程序等。

在选择菜单后的 WM_COMMAND 消息中,程序调用 DeleteBackGround 子程序先删除原有的位图,再调用 CreateBackGround 和 CreateClockPic 子程序产生新的背景位图和时钟位图,最后调用 InvalidateRect 函数产生 WM_PAINT 消息重新绘画客户区。

整个程序的结构采用图 7.1 中的 B 结构,也就是说窗口客户区的绘画代码和产生时钟位图的代码是分开的,程序中建立了两个位图当做缓冲数据,第一个位图是背景位图,在 `_CreateBackGround` 子程序中建立,只有在程序初始化,以及在菜单中选择了不同的背景和边框后才需要调用这个子程序,以便建立新的背景图片;第二个位图是要输出到屏幕的时钟位图,

它在 `_CreateClockPic` 子程序中建立，时钟位图是通过将背景位图拷贝过来，再根据当前时间画上指针得到的，程序在 `WM_TIMER` 消息中每秒重画一次新的时钟图片，并用 `InvalidateRect` 函数产生一个 `WM_PAINT` 消息将这个位图拷贝到屏幕上，当其他原因产生 `WM_PAINT` 消息时，程序并不从头开始产生背景位图和时钟位图等图片，而是直接从时钟图片中拷贝数据到屏幕上。

好了，接下来继续分析 `_CreateBackGround` 和 `_CreateClockPic` 子程序是如何对位图进行处理的。

7.3.2 创建和使用位图

所有绘图函数的操作对象都是“设备环境”，对位图操作也不例外。为了对位图使用 GDI 函数，需要使用 `CreateCompatibleDC` 函数为位图建立一个 DC，然后使用 `SelectObject` 函数将位图选入这个 DC 中，这样，所有对这个 DC 的绘图操作实际上就是在操作这个位图。每一个需要操作的位图都需要单独为它创建一个 DC。

程序中常常使用在资源中预定义的位图，但也有使用未初始化的位图的情况，如例子程序的背景位图和时钟位图一开始就是未初始化的，它们是程序开始运行后才被创建的。

为了建立一个未初始化的位图，可以使用以下的函数：

invoke	CreateCompatibleBitmap, hDC, dwWidth, dwHeight
mov	hBitmap1, eax ;方法 1
invoke	CreateBitmap, dwWidth, dwHeight, dwPlanes, dwBitsPerPel, NULL
mov	hBitmap2, eax ;方法 2

创建一个位图需要的参数是高度、宽度，以及颜色深度，要创建位图必须得知这些参数。使用 `CreateCompatibleBitmap` 创建位图的时候，参数中有一个 `hDC`，这是个参考 `hDC`，也就是说，新位图的颜色深度和 `hDC` 对应的“设备环境”的颜色深度相同（注意：有个 `hDC` 参数的意思并不是将创建的位图选入这个 `hDC` 中）。`CreateBitmap` 函数则直接在参数 `dwPlanes` 和 `dwBitsPerPel` 中指定了颜色深度。两个函数的 `dwWidth` 和 `dwHeight` 参数指定创建的位图的宽度和高度。

在例子程序的 `_CreateBackGround` 子程序中，为了建立背景图片和时钟图片，需要建立两个未初始化的位图及操作它们的 DC，所以程序一开始用 `GetDC` 函数获取主窗口的 `hDC` 来当做参考 DC，然后用 `CreateCompatibleDC` 函数建立了两个 DC（句柄放在全局变量 `hDcBack` 和 `hDcClock` 中），并用 `CreateCompatibleBitmap` 建立了两个位图（句柄放入 `hBmpBack` 和 `hBmpClock` 中），接下来用 `SelectObject` 将这两个位图选入新建的 `hDC` 中。

创建背景图片的过程中还要用到资源中的背景图片、边框图片和边框的遮掩图片，对于这些图片，程序用 `LoadBitmap` 函数装入，并使用 `CreateCompatibleDC` 为每个图片建立一个 DC。

对于不再使用的位图，要用 `DeleteObject` 函数将它们删除。所以在子程序的最后，使用 `DeleteObject` 函数将临时使用的位图句柄删除，并使用 `DeleteDC` 将操作这些位图的 `hDC` 删除。

 操作未初始化位图需要用到 CreateCompatibleDC 和 CreateCompatibleBitmap 函数, 初学者常犯的错误是用 CreateCompatibleDC 返回的 hDC 当做 CreateCompatibleBitmap 函数的参考 hDC, 这样的结果是建立的位图是单色的, 正确的做法是两个函数的参考 hDC 都使用窗口客户区的 hDC。

7.3.3 使用设备无关位图

设备无关位图简称为 DIB, 这在 5.3.1 节中已经有所介绍。DIB 一般是存放在磁盘上的以 bmp 为扩展名的位图文件, 使用 DIB 的关键是如何将 DIB 中的数据转化为一个内存中的位图并返回一个位图句柄。

bmp 文件的文件结构是这样定义的: 文件的开始是一个 BITMAPFILEHEADER 结构。这个结构定义如下:

BITMAPFILEHEADER	STRUCT		
bfType	WORD	?	;文件标识, 必须是“BM”
bfSize	DWORD	?	;文件长度
bfReserved1	WORD	?	;0
bfReserved2	WORD	?	;0
bfOffBits	DWORD	?	;位图像素数据在文件中的起始位置
BITMAPFILEHEADER	ENDS		

BITMAPFILEHEADER 结构的后面要么是 BITMAPCOREHEADER 结构, 要么是 BITMAPINFOHEADER 结构和索引色表, 这两种结构的定义如下:

BITMAPCOREHEADER	STRUCT		
bcSize	DWORD	?	;本结构长度
bcWidth	WORD	?	;位图宽度
bcHeight	WORD	?	;位图高度
bcPlanes	WORD	?	;位图的色平面数
bcBitCount	WORD	?	;位图的颜色深度
BITMAPCOREHEADER	ENDS		
BITMAPINFOHEADER	STRUCT		
biSize	DWORD	?	;本结构长度
biWidth	WORD	?	;位图宽度
biHeight	WORD	?	;位图高度
biPlanes	WORD	?	;位图的色平面数
biBitCount	WORD	?	;位图的颜色深度
biCompression	DWORD	?	;位图的压缩方式
biSizeImage	DWORD	?	;图形尺寸
biXPelsPerMeter	DWORD	?	;图形 x 方向分辨率, 单位是像素/米
biYPelsPerMeter	DWORD	?	;图形 y 方向分辨率, 单位是像素/米
biClrUsed	DWORD	?	
biClrImportant	DWORD	?	
BITMAPINFOHEADER	ENDS		

这两个数据结构主要包含了位图的一些参数, 在这些数据结构的后面, 就是位图的像素数据了, 整个 bmp 文件就由这 3 部分组成。

要使用 DIB，可以首先将整个文件读到内存中，然后从这些数据结构中得知位图的各种参数，最后使用 SetDIBitsToDevice 函数将位图数据复制到一个 hDC 中，如果这个 hDC 对应一个未初始化的位图，那么就相当于得到了包含磁盘 bmp 位图数据的位图句柄，并且可以在任何地方使用它。当然，在这以后可以将读入文件数据的内存释放掉。

SetDIBitsToDevice 函数的用法是：

```
invoke    SetDIBitsToDevice, hDC, xDest, yDest, \
          dwWidth, dwHeight, xSrc, ySrc, uStartScan, cScanLines, \
          lpvBits, lpbmi, fuColorUse
```

hDC 是目的 DC 的句柄，xDest 和 yDest 指定了位图复制到 hDC 的左上角位置，dwWidth 和 dwHeight 指定了要复制的宽度和高度，xSrc 和 ySrc 指定 DIB 中要复制的左上角位置，uStartScan 和 cScanLines 指定开始复制的扫描线和要复制的扫描线数，最后，lpvBits 指向 DIB 中的像素数据部分，lpbmi 指向 DIB 中的 BITMAPINFO 或 BITMAPCOREINFO 结构，fuColorUse 指定了 DIB 中数据的类型，用 DIB_RGB_COLORS 表示数据是 RGB 类型的。

子程序_CreateDIBitmap 分析一个 DIB 文件的参数并返回包含整个 DIB 位图数据的位图句柄，读者可以在任何地方使用这个位图句柄。子程序的输入参数_hWnd 用来获取参考 hDC 的窗口句柄，_lpFileData 是将 DIB 文件整个读入内存后的内存指针。代码如下：

```
_CreateDIBitmap    proc _hWnd, _lpFileData
                    local    @lpBitmapInfo, @lpBitmapBits
                    local    @dwWidth, @dwHeight
                    local    @hDc, @hBitmap

                    pushad
                    mov     @hBitmap, 0
                    mov     esi, _lpFileData
                    mov     eax, BITMAPFILEHEADER.bfOffBits [esi]
                    add     eax, esi
                    mov     @lpBitmapBits, eax
                    add     esi, sizeof BITMAPFILEHEADER
                    mov     @lpBitmapInfo, esi
                    .if     BITMAPINFO.bmiHeader.biSize [esi] == sizeof BITMAPCOREHEADER
                        movzx    eax, BITMAPCOREHEADER.bcWidth [esi]
                        movzx    ebx, BITMAPCOREHEADER.bcHeight [esi]
                    .else
                        mov     eax, BITMAPINFOHEADER.biWidth [esi]
                        mov     ebx, BITMAPINFOHEADER.biHeight [esi]
                    .endif
                    mov     @dwWidth, eax
                    mov     @dwHeight, ebx

;*****
; 建立空的 Bitmap Object
;*****
                    invoke    GetDC, _hWnd
                    push    eax
                    invoke    CreateCompatibleDC, eax
                    mov     @hDc, eax
                    pop     eax
```

```

        push eax
        invoke CreateCompatibleBitmap, eax, @dwWidth, @dwHeight
        mov     @hBitmap, eax
        invoke SelectObject, @hDc, @hBitmap
        pop     eax
        invoke ReleaseDC, hWinMain, eax
;*****
; 将文件内容设置到建立的 Bitmap 中
;*****
        invoke SetDIBitsToDevice, @hDc, 0, 0, @dwWidth, @dwHeight, \
            0, 0, 0, @dwHeight, \
            @lpBitmapBits, @lpBitmapInfo, DIB_RGB_COLORS
        .if     eax == 0
            invoke DeleteObject, @hBitmap
            mov     @hBitmap, 0
        .endif
        invoke DeleteDC, @hDc
        popad
        mov     eax, @hBitmap
        ret

_CreateDIBitmap endp

```

_CreateDIBitmap 子程序首先分析 DIB 文件的数据，确定 BITMAPFILEHEADER 后面的数据结构是 BITMAPINFO 还是 BITMAPCOREINFO，并从结构中获取位图的高度和宽度，然后建立一个未初始化的位图，并用 SetDIBitsToDevice 函数将位图数据拷贝到这个位图中，最后将位图句柄返回以供后面使用。

7.4 块传送操作

除了 7.2 节中的绘图函数，块传送函数也是重要的图形操作函数。块传送指把源位置中的数据块按照指定的方式传送到目的位置中。把内存中的位图复制到窗口客户区，以及在不同的 DC 中复制图形数据都要用到块传送操作，块传送完成的工作就相当于图形之间的拷贝工作。块传送函数有 PatBlt, BitBlt, MaskBlt, PlgBlt, TransparentBlt 和 StretchBlt 等。

7.4.1 块传送方式

与 7.2.4 节中介绍的绘图函数的 ROP 操作类似，块传送函数也是可以用 ROP 码来定义的传送方式，但块传送函数的 ROP 码定义不同于 7.2.4 节中的 ROP 码，因为这里涉及的操作对象更多。

表 7.6 块传送函数中使用的 ROP 码

ROP 码	十六进制数值	新像素点算法
BLACKNESS	00000042h	全部为 0
DSTINVERT	00550009h	Not 目标像素
MERGECOPY	00c000cah	画刷 and 源像素

MERGEPAINT	00bb0226h	(not 源像素) or 目标像素
NOTSRCCOPY	00330008h	Not 源像素
NOTSRCERASE	001100a6h	not (源像素 or 目标像素)
PATCOPY	00f00021h	画刷
PATINVERT	005a0049h	画刷 xor 目标像素
PATPAINT	00fb0a09h	画刷 or (not 源像素) or 目标像素
SRCAND	008800c6h	源像素 and 目标像素
SRCCOPY	00cc0020h	源像素
SRCERASE	00440328h	源像素 and (not 目标像素)
SRCINVERT	00660046h	源像素 xor 目标像素
SRCPAINT	00ee0086h	源像素 or 目标像素
WHITENESS	00ff0062h	全部为 1

块传送的 ROP 码是一个 32 位的整数，对应的操作涉及 3 种原始数据：源像素、目标像素和画刷，块传送操作的结果是目标像素的数据被 3 种原始数据的计算结果替换掉。

并不是每一种 ROP 码都要用到全部 3 种原始数据，有的甚至连 1 种也用不到，例如，全黑或者全白的 ROP 码。块传送函数使用的 ROP 码总共有 256 种，它们是 3 种原始数据进行不同位操作（取反、与、或和异或）的组合，但有些 ROP 码对应的操作结果实在是太难想象了，比如，ROP 码 0e20746 对应的操作是((目标像素 xor 画刷) and 源像素) xor 目标像素)，凭这个算式的确比较难以想象最后得到的位图是什么样子的！由于在实际使用中很多算法组合也并不是那么有用，所以 Windows 只对 15 种最常用的 ROP 码定义了预定义的助记代码。

如表 7.6 所示，对于这些 ROP 码，在程序中可以直接使用助记码，对于表中没有列出的 ROP 码，可以直接使用十六进制数值。

7.4.2 块传送函数

1. PatBlt 函数

PatBlt 函数完成的是“图案块传送”的功能，即“**P**attern **B**lock **T**ransfer”。使用的方法是：

invoke	PatBlt, hDC, xDest, yDest, dwWidth, dwHeight, dwROP
--------	---

这个函数将当前画刷的图案拷贝到 hDC 中以 (xDest, yDest) 为左上角坐标，dwWidth 为宽度，dwHeight 为高度的区域中，传送的方式由 dwROP 中的 ROP 码指定。PatBlt 函数的功能和矩形填充函数 FillRect 与 InvertRect 等是很像的，但它包含了它们的全部功能，如 ROP 码指定 DSTINVERT，那么 PatBlt 的功能就相当于 InvertRect 函数；ROP 码指定为 PATCOPY 的时候，PatBlt 的功能相当于 FillRect 函数。

在 BmpClock.asm 的 _CreateBackGround 子程序中，当建立背景图片的时候，就是用 PATCOPY 方式的 PatBlt 函数用资源中的背景图片填充整个时钟背景的。

在所有的 ROP 码中，可以用在 PatBlt 函数中的只有一部分，所有算法中包含源像素的 ROP 码在 PatBlt 函数中都不能使用，因为 PatBlt 函数只涉及当前画刷和目标像素，并没有一个“源像素”，所以对于这个函数来说，表 7.6 中的 ROP 码中只有 BLACKNESS, WHITENESS, DSTINVERT, PATINVERT 和 PATCOPY 是可用的。

2. BitBlt 函数

PatBlt 函数能完成的工作 BitBlt 函数都能完成，BitBlt 是“数据块传送”的意思，即“**Bit Block Transfer**”。BitBlt 函数的用法是：

```
invoke    BitBlt, hDcDest, xDest, yDest, dwWidth, dwHeight, \
          hDcSrc, xSrc, ySrc, dwROP
```

这个函数将源 hDcSrc 中以 (xSrc, ySrc) 为左上角的一个矩形区域传送到目标 hDcDest 中以 (xDest, yDest) 为左上角的地方去，矩形的宽度为 dwWidth，高度为 dwHeight，当然目标 DC 中的最后结果是由 dwROP 中的 ROP 码定义的源、目标和画刷三者数据的组合。

灵活使用 ROP 码可以实现很多的功能，比如，在一个背景图片上叠加一个非矩形的位图，游戏程序中人物在背景上面的移动就是这样的一个例子。BmpClock 程序中也实现了类似的功能——读者可以注意到，程序可以自由更换背景和边框，但是边框是中空的，它相当于以一个不规则的图形叠加在背景上面，图 7.10 示范了实现的方法。

分析一下 BmpClock.asm 中的 _CreateBackGround 子程序就可以发现，程序用到了资源中的 Back1.bmp, Mask1.bmp 和 Circle1.bmp 这 3 个图片（在图 7.10 中以 A, B, C 来表示），子程序将 3 个图片装入内存后，创建了 3 个 DC 来存取它们，对应的 DC 句柄分别放在 @hBmpBack, @hBmpMask 和 @hBmpCircle 中。

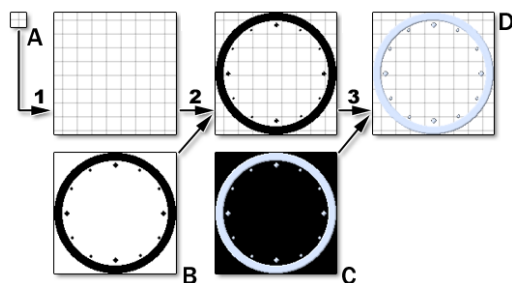


图 7.10 在背景上叠加不规则图形的方法

好了！现在的任务是将图片 C 中需要的部分（非黑色部分）透明叠放在以图片 A 形成的背景上，得到时钟背景图片 D。继续做准备工作：为图片 D 建立一个未初始化位图和内存 DC，DC 句柄存放在 hDcBack 中。

如图 7.10 中的步骤 1 所示，首先，程序用 CreatePatternBrush 建立以图片 A 为图案的画刷，用 PatBlt 函数以这个画刷填充整个图片：

```
invoke    CreatePatternBrush, @hBmpBack
push     eax
invoke    SelectObject, hDcBack, eax
```

invoke	PatBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, PATCOPY
pop	eax
invoke	DeleteObject, eax

现在如果直接将图片 C 拷贝上去, 虽然需要的部分是拷贝上去了, 但是图片 C 中的黑色部分也会覆盖全部的背景像素, 为了让黑色部分的背景像素保持不变, 应该使用 or 操作, 因为黑色的颜色值为 0, 任何数据和 0 进行 or 操作将保持不变, 查看 ROP 码可以发现, SRCPAINT 使用的是 or 操作, 所以可以使用 SRCPAINT 操作码进行 BitBlt 操作。

但还有个问题: 图片 C 中的非黑色部分如果也用 or 操作绘画到背景上, 那么经过和背景像素的 or 操作后就不是原来的颜色值了, 为了让这部分不变, 解决的办法是预先将背景中对应的部分先绘制成黑色, 这样对应图片 C 中的黑色部分将保持背景颜色, 而非黑色部分将使用图片 C 中的像素。遮掩图片 B 就是这样用的, 它是一个黑白两色的图片, 黑色部分是图片 C 中要在背景上“镂空”的部分, 在步骤 2 中, 程序使用下列语句将图片 B 用 SRCAND 操作码绘制到背景上:

invoke	BitBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, @hDcMask, 0, 0, SRCAND
--------	---

查表 7.6 可以发现, SRCAND 进行源像素和目标像素的 and 操作, 任何数和 1 进行 and 将保持不变, 和 0 进行 and 将变成 0, 这样背景中对应图片 B 中的白色部分将保持不变, 对应图片 B 中的黑色部分将被“镂空”。

接下来就是最后的步骤 3 了:

invoke	BitBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, @hDcCircle, 0, 0, SRCPAINT
--------	---

程序用 SRCPAINT 操作码将图片 C 和已经镂空的背景进行 or 操作, 得到的结果就是将由遮掩图片 B 指定的图片 C 中的不规则区域画到了背景上面。

为了简化起见, BmpClock 程序中的遮掩图片是预先设计好的, 实际使用中也可以通过扫描源位图中的像素位, 找出背景颜色并动态生成一个遮掩位图, 虽然这样可能对速度有一些影响, 但灵活性要高得多。

在游戏程序中, 将一个不规则的图形如人或物体等图形叠加到背景上面使用的就是这样的技术。

3. MaskBlt 函数

MaskBlt 函数允许在一个图片中对不同的部分以不同的 ROP 码进行操作, 它的语法是:

invoke	MaskBlt, hDcDest, xDest, yDest, dwWidth, dwHeight, \ hDcSrc, xSrc, ySrc, hMaskBmp, xMask, yMask, dwROP
--------	---

它和 BitBlt 的不同之处是多了个遮掩图片句柄 hMaskBmp (注意: 是位图句柄而不是 DC 句柄), 以及 hMaskBmp 的开始坐标。

MaskBlt 函数同样是将 hDcSrc 以 (xSrc, ySrc) 为左上角的矩形区域以指定 ROP 码操作方式传送到 hDcDest 中以 (xDest, yDest) 为左上角的矩形区域中, 矩形的宽度和高度由 dwWidth

和 dwHeight 指定，函数的特殊之处是可以指定两个 ROP 码，传送时使用哪个 ROP 码要参考遮掩图片，参考的位置从遮掩图片的 (xMask, yMask) 坐标开始。

hMaskBmp 指定了一幅黑白位图，如果位图中对应位置的像素为黑（即为 0），那么使用背景 ROP 码，如果对应位置的像素为白（即为 1），那么使用前景 ROP 码。注意：遮掩位图一定要是黑白两色的，如果使用其他颜色深度的位图，那么函数调用将会失败。

读者一定有个问题：参数中只有一个 dwROP 参数，如何指定两个 ROP 码？这正是这个函数比较费解的地方，实际上 dwROP 参数是两个 ROP 码的组合，组合的算法是：

((背景 ROP 码 shl 8) & 0ff000000h) or 前景 ROP 码

灵活使用这个函数可以带来很多的方便，比如在 _CreateBackGround 子程序中，以“;\$\$\$\$”为注释之间的 8 句代码完全可以改成下面的 4 句：

```
invoke CreatePatternBrush, @hBmpBack
invoke SelectObject, hDcBack, eax
invoke DeleteObject, eax
invoke MaskBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, \
    @hDcCircle, 0, 0, @hBmpMask, 0, 0, \
    ((SRCCOPY shl 8) and 0ff000000h) or PATCOPY
```

相比之下少了一个 PatBlt 和一个 BitBlt 调用。因为程序的要求是遮掩图片中黑色的部分使用边框图片，而白色部分使用背景，所以将背景 ROP 设置为 SRCCOPY 就可以将需要的边框部分复制过去，而前景 ROP 使用 PATCOPY 就免除了使用 PatBlt 填充背景的步骤。当然，使用前需要把 Mask1. bmp 和 Mask2. bmp 重新保存成黑白色，因为现在这两个位图文件是 256 色格式的。

如果 hMaskBmp 不指定（参数写为 NULL），那么 MasmBlt 函数就相当于 BitBlt 函数，只不过 ROP 是使用 dwROP 中指定的前景 ROP 码。

4. PlgBlt 函数

PlgBlt 实现平行四边形旋转传送操作（Parallelogram Block Transfer），它复制一幅位图，同时将其转换成一个平行四边形，所以利用它可对位图进行旋转处理。PlgBlt 函数的使用语法是：

```
invoke PlgBlt, hdcDest, lpPoint, \
    hdcSrc, xSrc, ySrc, dwWidth, dwHeight, \
    hBmpMask, xMask, yMask
```

这个函数与 ROP 码无关，但它同样指定了一个单色的遮掩位图，遮掩位图中为 1 的像素对应的位置会被传送，为 0 的像素不被传送。(xSrc, ySrc) 指定了源 DC 中需要传送的矩形的左上角，dwWidth 和 dwHeight 指定宽度和高度，这个矩形将被旋转后传送到目的 DC 中去，旋转后的平行四边形位置由 lpPoint 指定的 POINT 结构阵列指出。

lpPoint 是一个指针，指向含有 3 个 POINT 结构的内存中（这种使用 POINT 结构数组的方法在 Polyline 中已经使用过），其中第一个点对应于一个平行四边形的左上角位置，第二个点代表右上角位置，第三个点代表右下角的位置，不需要第四个点是因为它可以从上面 3 个点的坐标推导出来。

5. StretchBlt 函数

StretchBlt 函数实现像素的缩放功能，它的语法是：

```
invoke    StretchBlt, hDcDest, xDest, yDest, dwWidthDest, dwHeightDest, \
          hDcSrc, xSrc, ySrc, dwWidthSrc, dwHeightSrc, dwROP
```

这个函数将源 hDcSrc 中以 (xSrc, ySrc) 为左上角，dwWidthSrc 和 dwHeightSrc 为宽度和高度的矩形以 dwROP 指定的方式传送到目标 hDcDest 中去，目标位置是 (xDest, yDest)，新的矩形区域大小为 dwWidthDest 和 dwHeightDest，如果源 DC 中的矩形大小和目标 DC 中的不一样，函数会将像素数据自动缩放。

但是 StretchBlt 对像素的缩放办法仅仅是删除多余的像素（从大到小）或者重复像素（从小到大），并不像一些图形处理软件一样进行插值计算，所以缩放的效果并不好，笔者建议如果能够不用这个函数就不要去用它，除非对图形的质量并没有要求。

6. TransparentBlt 函数

还有一个函数可以方便地实现不规则区域的像素拷贝操作，那就是 TransparentBlt 函数，它的用法如下：

```
invoke    TransparentBlt, hDcDest, xDest, yDest, dwWidthDest, dwHeightDest, \
          hDcSrc, xSrc, ySrc, dwWidthSrc, dwHeightSrc, crTransparent
```

可以看出，它和 StretchBlt 函数的参数很像，也有 dwWidthSrc 和 dwHeightSrc 参数，难道这个函数也可以缩放吗？答案是肯定的，它也可以进行像素缩放。TransparentBlt 函数的最后一个参数指定了一个透明色，源 DC 指定的矩形区域中和这个颜色相同的像素不会被拷贝，所以，BmpClock 程序中的下列两个语句：

```
invoke    BitBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, @hDcMask, 0, 0, SRCAND
invoke    BitBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, \
          @hDcCircle, 0, 0, SRCPAINT
```

完全可以用下面这一句来代替：

```
invoke    TransparentBlt, hDcBack, 0, 0, CLOCK_SIZE, CLOCK_SIZE, \
          @hDcCircle, 0, 0, CLOCK_SIZE, CLOCK_SIZE, 0
```

这样甚至连遮掩图片 Mask1. bmp 和 Mask2. bmp 都可以省掉了。

当然，这个函数的传送方式使用拷贝方式，如果需要用到 ROP 码，那么只有使用其他函数了。由此可见，各种块传送函数都有它们的优缺点，在实际应用中，最好的办法就是根据实际情况灵活使用。

TransparentBlt 函数并不包含在 Gdi32. dll 中，而是在 Msimg32. dll 中，所以使用时注意在源程序头部加上下面的包含语句：

```
include    Msimg32. inc
includelib Msimg32. lib
```

7.5 区域和路径

如果读者用过 PhotoShop 绘图软件，就一定记得它有“选择区域”，以及“路径”的概念，区域用来选定一个范围，以便对指定的范围进行某种操作；而路径相当于用虚拟的线条去进行“圈地运动”，虽然路径圈出来的看上去也是一个区域，但路径记录的是“圈地”用的线条而不是圈出来的地。

7.5.1 使用区域

1. 创建区域

首先来看如何建立一个区域，GDI 中可以用下列区域建立函数来创建区域：

(1) invoke	CreateRectRgn, dwLeft, dwTop, dwRight, dwBottom
(2) invoke	CreateEllipticRgn, dwLeft, dwTop, dwRight, dwBottom
(3) invoke	CreateEllipticRgnIndirect, lpRect
(4) invoke	CreatePolygonRgn, lpPoint, iCount, iPolyFillMode
(5) invoke	CreateRoundRectRgn, dwLeft, dwTop, dwRight, dwBottom, \
	dwWidthEllipse, dwHeightEllipse

这些函数分别创建矩形（1）、椭圆形（2）和（3）、多边形（4）和圆角矩形（5）的区域，参数中坐标值的含义和绘画填充区域的函数 Rectangle, Ellipse, Polygon 与 RoundRect 的参数定义是相同的。

CreateRectRgn 函数的参数指定了左上和右下两个对角的坐标。CreateEllipticRgn 的参数指定了一个矩形，产生的椭圆区域和这个矩形是相切的。

CreateEllipticRgnIndirect 函数同样建立椭圆区域，但指定与椭圆相切的矩形是由一个 RECT 结构定义的。

CreatePolygonRgn 建立一个多边形区域。lpPoint 指向一系列的 POINT 结构，iCount 指定了点的数量，iPolyFillMode 参数就是 SetPolyFillMode 函数使用的参数：ALTERNATE 或 WINDING，结果的不同之处相当于表 7.4 中 Polygon 函数的两种结果的区别。

如果创建区域成功的话，这些函数返回一个区域句柄 hRgn，区域和画笔、画刷等一样，都是 GDI 的对象，如果不再使用一个区域，需要用 DeleteObject 将它删除。

2. 合并区域

仅使用上面的函数可能用途不是很大，但是将不同形状的区域按照各种方式合并起来，用处就大了，可以因此定义出很复杂的区域。要合并区域可以使用函数：

invoke	CombineRgn, hDestRgn, hSrcRgn1, hSrcRgn2, dwCombine
--------	---

该函数将 hSrcRgn1 和 hSrcRgn2 两个区域合并起来放入 hDestRgn 指定的区域句柄中，但 hDestRgn 并不是新生成的，它必须是一个已经存在的区域句柄，当函数执行后，hDestRgn 中原来的区域会被破坏并替换成新合并成的区域，但可以对 hDestRgn 和 hSrcRgn1 使用同一个句柄，这样就相当于把 hSrcRgn2 合并到 hSrcRgn1 中去。

dwCombine 指定了合并的方式，它可以是以下的取值：

- RGN_AND——新区域是两个区域的共同部分。
- RGN_COPY——新区域是 hSrcRgn1 中的区域。
- RGN_DIFF——新区域是 hSrcRgn1 区域除去 hSrcRgn2 中的部分。
- RGN_OR——新区域是两个区域的叠加。
- RGN_XOR——新区域是两个区域的叠加除去共同部分。

CombineRgn——函数的返回值代表新区域的形状大致是什么样子的，它可能是以下值：

- NULLREGION——新区域是空区域。
- SIMPLEREGION——新区域是个简单形状（可以用上面的单个函数创建）。
- COMPLEXREGION——新区域是个复杂的形状。
- ERROR——函数执行出错，没有区域被创建。

3. 区域的用途

区域主要可以用在两个地方：建立特殊形状的窗口和对绘画区域进行裁剪。

使用 SetWindowRgn 函数可以使窗口的形状由区域指定，如 BmpClock 时钟程序是圆形的，当把时钟移动到其他窗口上面的时候，它的四角并不覆盖住下层窗口，这就是因为程序中有下面的代码：

```

invoke    CreateEllipticRgn, 0, 0, CLOCK_SIZE+1, CLOCK_SIZE+1
push     eax
invoke    SetWindowRgn, hWinMain, eax, TRUE
pop      eax
invoke    DeleteObject, eax

```

在这几句代码中，CreateEllipticRgn 函数建立了一个圆形的区域并在 eax 中返回区域句柄，SetWindowRgn 函数根据这个区域设置时钟窗口的形状。如果建立了一个很复杂的区域，那么窗口的形状同样会很复杂。SetWindowRgn 的最后一个参数为 TRUE，表示设置窗口形状后 Windows 要发送一个 WM_PAINT 消息将窗口重画。

由于 Windows 对使用的区域保存一个拷贝，所以程序在调用 SetWindowRgn 函数后就可以使用 DeleteObject 把区域删除掉，并不需要在退出时才去删除。

另外，区域可以用来对绘画区域进行裁剪，任意使用下面的两条语句之一：

```

invoke    SelectObject, hDC, hRgn
invoke    SelectClipRgn, hDC, hRgn

```

那么以后在 hDC 上使用绘图函数的话，只有 hRgn 指定的区域中的点才会被绘画，对裁剪区域外的绘图将被忽略。同样，Windows 会对选入 DC 的区域建立一个拷贝，如果以后不需要这个区域了，那么在函数执行后，可以马上用 DeleteObject 函数把它删除掉。

7.5.2 使用路径

1. 创建路径

路径并不是 GDI 对象，它并没有一个句柄，Windows 对每个 DC 在内部保存一个路径，每次新开始建立一个路径，原有的路径就会被破坏掉。如果要建立一个路径，可以使用 BeginPath 函数：

invoke	BeginPath, hDC
--------	----------------

调用了这个函数以后，对 hDC 使用画线函数所画的线条都会被当做路径记录，使用画线函数画出来的线条可能是不连续的，比如调用多次的 LineTo 函数，最后一点和开始一点不同，这时需要使用 CloseFigure 函数将路径封闭起来：

invoke	CloseFigure, hDC
--------	------------------

CloseFigure 函数从最后一点到第一点画一条直线把路径封闭起来。Windows 允许创建多个子路径，封闭前面一条路径以后，可以继续“圈”出和前面路径不相连的另一条路径。

最后，使用 EndPath 函数结束创建路径：

invoke	EndPath, hDC
--------	--------------

图 7.11 示范了一次创建路径的过程，在第 1 步调用 BeginPath 和第 11 步调用 EndPath 之间，用 LineTo 函数和 Rectangle 画出了包含两个三角形和一个矩形的路径，其中两个三角形用 CloseFigure 去封闭。

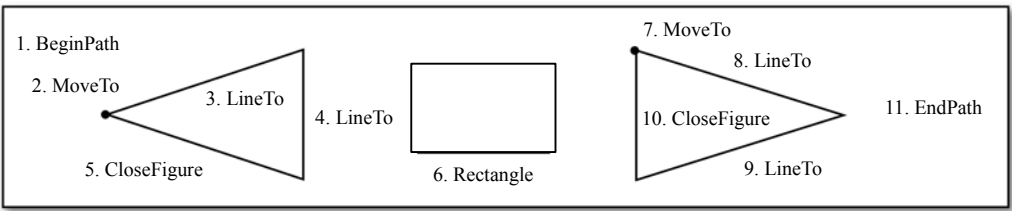


图 7.11 路径创建过程

2. 使用路径

创建了路径以后，可以进行下面的操作。

首先，可以对路径进行画线操作，或者对路径围起来的区域进行填充操作：

invoke	StrokePath, hDC	; (1)
invoke	FillPath, hDC	; (2)
invoke	StrokeAndFillPath, hDC	; (3)

第（1）个函数沿着路径用当前画笔绘画线条，第（2）个函数使用当前画刷填充路径围起来的区域，第（3）个函数既使用当前画笔绘画边线也使用当前画刷填充中间区域。当执行了任何一个函数的时候，路径都会被破坏掉。实际上，这些函数完成的功能就相当于 7.2.3 节中的画线和填充函数，那么为什么要这样大动干戈呢？惟一的好处就是用这种方法可以操作很复

杂的形状，因为定义路径时可以使用任何画线函数，包括画弧与画贝塞尔曲线函数等，而用普通的填充函数是无法填充出一个由贝塞尔曲线围成的区域的。

路径的另一个用途是定义一个复杂形状的区域，可以使用下面的函数将路径转化成区域：

invoke	PathToRegion, hDC
mov	hRgn, eax

理由是同样的，因为用区域创建函数创建出来的只能是椭圆、矩形、多边形等形状的组合，用先创建路径再转化成区域的方法可以定义形状复杂得多的区域。同样，执行了 PathToRegion 函数以后，原有的路径定义就会被破坏掉。

Windows 操作系统为一些常用功能提供了一些通用对话框（Common Dialog Box），比如，在不同的应用程序中进行打开文件、选择字体、选择颜色等操作时，不同程序显示的对话框的模样都是一样的。另外，把同样的应用程序放到不同版本的 Windows 下执行就会发现，这些对话框会随着操作系统版本的不同而不同，如图 8.1 所示，选择同样的“打开”文件菜单项时，在 Windows XP 下显示的对话框是左边的样子，而在 Windows 98 下显示的是右边的样子，但程序中并没有为不同版本的操作系统设计不同的对话框。造成这些现象的原因就是这些对话框是操作系统提供的，实现对话框的代码包括在 Comdlg32.dll 库文件中，由于不同版本的 Comdlg32.dll 在设计上可能有所不同，所以不同版本 Windows 下的对话框会有所不同。

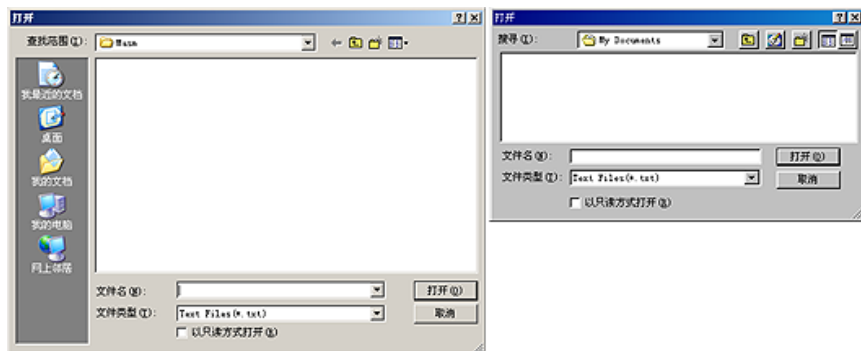


图 8.1 不同操作系统下的“打开”文件对话框

8.1 通用对话框简介

通用对话框函数由 Comdlg32.dll 提供，在使用之前需要在源程序中包含相应的 include 和 includelib 语句：

```
include    comdlg32.inc
includelib comdlg32.lib
```

Windows 提供多种通用对话框，每种通用对话框都使用一个专用的函数来创建和显示，另

Comdlg32.dll 中提供的通用对话框如表 8.1 所示, 表中还包括创建这些对话框使用的函数, 以及数据结构名称。

表 8.1 通用对话框列表

通用对话框	使用函数	使用数据结构
选择颜色	ChooseColor	CHOOSECOLOR
查找字符串	FindText	FINDREPLACE
替换字符串	ReplaceText	FINDREPLACE
选择字体	ChooseFont	CHOOSEFONT
打开文件	GetOpenFileName	OPENFILENAME
保存文件	GetSaveFileName	OPENFILENAME
页面设置	PageSetupDlg	PAGESETUPDLG

CommDlg.asm 和资源脚本文件 CommDlg.rc。CommDlg.asm 文件的内容如下:

```

.386
.model flat, stdcall
option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include      windows.inc
include      user32.inc
includelib   user32.lib
include      kernel32.inc
includelib   kernel32.lib
include      Comdlg32.inc
includelib   Comdlg32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Equ 等值定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN      equ      1000
DLG_MAIN      equ      1000
IDM_MAIN      equ      1000
IDM_OPEN      equ      1101
IDM_SAVEAS    equ      1102
IDM_PAGESETUP equ      1103
IDM_EXIT      equ      1104
IDM_FIND      equ      1201
IDM_REPLACE   equ      1202
IDM_SELFONT   equ      1203
IDM_SELCOLOR  equ      1204

```

哭

246

```
;))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

248

资源脚本文件 CommDlg.rc 的内容如下:

249

在资源脚本文件中定义了一个对话框用做主窗口，同时定义了一个菜单用来选择各种通用对话框。下面结合这个例子说明各种通用对话框的用法。

8.2.1 “打开”文件和“保存”文件对话框

显示“打开”文件对话框的函数是 `GetOpenFileName`，显示“保存”文件对话框的函数是 `GetSaveFileName`。这两个对话框可以让用户选择驱动器、目录，以及一个文件名（打开文件对话框还允许选择多个文件），但这两个对话框并不对文件进行任何操作，也就是说，它们仅给用户提供一个统一的界面来“选择”文件名，获取文件名以后，对文件的打开、读写等操作还需要程序自己解决。

invoke	GetOpenFileName, lpofn	;显示打开文件对话框
invoke	GetSaveFileName, lpofn	;显示保存文件对话框

250

OPENFILENAMEA	STRUCT	
lStructSize	DWORD	? ;结构的长度, 用户填写
hWndOwner	DWORD	? ;所属窗口, 可以为 NULL
hInstance	DWORD	? ;
lpstrFilter	DWORD	? ;文件筛选字符串
lpstrCustomFilter	DWORD	? ;
nMaxCustFilter	DWORD	? ;
nFilterIndex	DWORD	? ;
lpstrFile	DWORD	? ;全路径的文件名缓冲区
nMaxFile	DWORD	? ;文件名缓冲区长度
lpstrFileName	DWORD	? ;不包含路径的文件名缓冲区
nMaxFileName	DWORD	? ;文件名缓冲区长度
lpstrInitialDir	DWORD	? ;初始目录
lpstrTitle	DWORD	? ;对话框标题
Flags	DWORD	? ;标志
nFileOffset	WORD	? ;文件名在字符串中的起始位置
nFileExtension	WORD	? ;扩展名在字符串中的起始位置
lpstrDefExt	DWORD	? ;默认扩展名
lCustData	DWORD	? ;
lpfnHook	DWORD	? ;
lpTemplateName	DWORD	? ;
OPENFILENAMEA	ENDS	
OPENFILENAME	equ	<OPENFILENAMEA>

结构中一些重要的字段含义如下。

- lpstrFilter——指定文件名筛选字符串, 该字段决定了对话框中“文件类型”下拉式列表框中的内容, 字符串可以由多组内容组成, 每组包括一个说明字符串和一个筛选字符串, 字符串的最后用两个 0 结束。如下面的字符串将在列表框中显示两项内容, 选择不同项目的时候分别列出“*.txt”文件或者所有文件“*.*”:

```
'Text Files(*.txt)', 0, '*.txt', 0, 'All Files(*.*)', 0, '*.*', 0, 0
```

筛选字符串中也可以同时指定多个扩展名, 中间用分号隔开, 如 '*.txt;*.doc'。

- lpstrFile——指向一个包含文件名的缓冲区。如果这个缓冲区中已经包含了一个文件名, 那么对话框初始化的时候将显示这个文件名。当用户选择了一个文件的时候, 函数在这里返回新的文件名。
- nMaxFile——指定 lpstrFile 参数指向的缓冲区的长度。
- lpstrFileName——指向一个缓冲区, 用来接收用户选择的不含路径的文件名。这个字段可以为 NULL。
- nMaxFileName——指明 lpstrFileName 参数指向的缓冲区的长度。
- lpstrInitialDir——对话框的初始化目录, 这个字段可以为 NULL。
- lpstrTitle——指向自定义的对话框标题, 如果这个字段是 NULL, 那么“打开”对话框和“保存”对话框的默认标题是“打开”和“另存为”。
- nFileOffset——返回文件名字符串中文件名的起始位置, 如当用户选择了文件“c:\dir1\file.ext”时, 这里将返回 8。

- `nFileExtension`——返回文件名字符串中扩展名的起始位置，同样是上面的字符串，这里返回 13。如果文件名的最后一个字符是“.”，这里返回 0，表示文件没有扩展名，这个字段和 `nFileOffset` 字段为分析文件名提供了方便。
- `lpstrDefExt`——指定默认扩展名，如果用户输入了一个没有扩展名的文件名，那么函数会自动加上这个默认扩展名。
- `Flags` 字段——该标志字段决定了对话框的不同行为，它可以是一些取值的组合。下面是一些比较重要的标志：
 - `OFN_ALLOWMULTISELECT`——允许同时选择多个文件名。
 - `OFN_CREATEPROMPT`——如果用户输入了一个不存在的文件名，对话框向用户提问“是否建立文件”。
 - `OFN_FILEMUSTEXIST`——用户只能选择一个已经存在的文件名，使用这个标志的时候必须同时使用 `OFN_PATHMUSTEXIST` 标志。
 - `OFN_HIDEREADONLY`——对话框中不显示“以只读方式打开”复选框。
 - `OFN_OVERWRITEPROMPT`——在“保存”文件对话框中使用的时候，当选择一个已存在的文件时，对话框会提问“是否覆盖文件”。
 - `OFN_PATHMUSTEXIST`——用户输入文件名时，路径必须存在。
 - `OFN_READONLY`——对话框中的“以只读方式打开”复选框初始化的时候处于选中状态。

调用显示“打开”或“保存”文件对话框函数时，函数会停留直到对话框关闭为止，当用户单击了对话框中的“确定”按钮时，函数返回 TRUE，用户单击“取消”按钮退出时，函数返回 FALSE，程序可以由此判断是否需要继续进行打开或保存文件的操作。具体的代码可以参考例子中的 `_SaveAs` 和 `_OpenFile` 子程序。

8.2.2 字体选择对话框

“字体”通用对话框如图 8.2 所示，对话框列出了系统中安装的字体，用户可以在上面选择字体名称，同时可以选择字体大小、颜色，以及一些效果如斜体、粗体、删除线或下划线等，显示选择“字体”对话框的函数是 `ChooseFont`：

invoke `ChooseFont, lpcf`

`lpcf` 指向一个 `CHOOSEFONT` 结构，这个结构是这样定义的：

CHOOSEFONT	STRUCT	
<code>lStructSize</code>	DWORD	? ;结构长度
<code>hwndOwner</code>	DWORD	? ;所属窗口
<code>hdc</code>	DWORD	? ;
<code>lpLogFont</code>	DWORD	? ;指向一个 LOGFONT 结构
<code>iPointSize</code>	DWORD	? ;选择的字体大小
<code>Flags</code>	DWORD	? ;标志
<code>rgbColors</code>	DWORD	? ;选择的字体颜色

lCustData	DWORD	?
lpfnHook	DWORD	?
lpTemplateName	DWORD	?
hInstance	DWORD	?
lpszStyle	DWORD	?
nFontType	WORD	?
Alignment	WORD	?
nSizeMin	DWORD	?
nSizeMax	DWORD	?
CHOOSEFONT	ENDS	

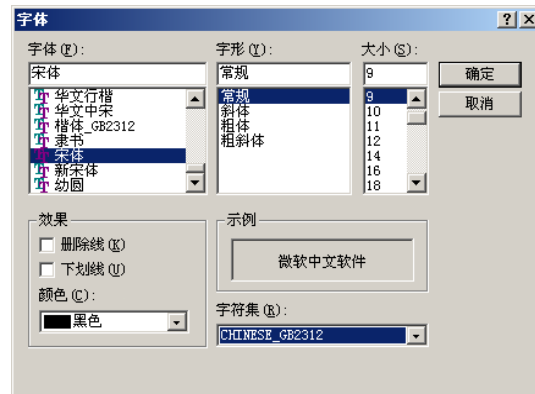


图 8.2 “字体选择”对话框

结构中一些重要的字段含义如下：

- hDC——当 Flags 字段中指定 CF_PRINTERFONTS 标志时，它是打印机的 DC 句柄。
- lpLogFont——指向一个包含 LOGFONT 结构的缓冲区。LOGFONT 结构可以用来指定字体的名称和属性。如果 Flags 字段中指定 CF_INITTOLOGFONTSTRUCT 标志的话，对话框将根据这个结构初始化对话框，函数也在这里返回用户选择的字体名称。
- iPointSize——返回用户选择的字体大小，单位是 1/10 磅。
- rgbColors——如果 Flags 字段的 CF_EFFECTS 标志被设置，对话框将根据这个数值初始化“颜色”下拉式列表框。另外，函数返回时在这里返回用户选择的字体颜色。
- nFontType——返回用户选择的字体是属于哪一类的，可能返回的值有 BOLD_FONTTYPE，ITALIC_FONTTYPE，PRINTER_FONTTYPE，REGULAR_FONTTYPE 和 SCREEN_FONTTYPE 等。

另一个关键的字段是 Flags 字段，Flags 字段的初始值决定了对话框的不同行为，函数返回的时候也会在这里返回一些用户的选择，它可以是下面取值的组合：

- CF_BOTH——对话框同时列出打印机字体和屏幕字体。
- CF_TTONLY——对话框只列出 TrueType 字体。
- CF_EFFECTS——对话框中显示“效果”复选框。

- CF_FIXEDPITCHONLY——对话框的字体列表中只显示等宽字体。
- CF_LIMITSIZE——对话框显示的字体尺寸限于 nSizeMin 和 nSizeMax 字段指定的数值之间。
- CF_NOSTYLESEL——对话框中不显示“字形”组合列表框。
- CF NOSIZESEL——对话框中不显示“大小”组合列表框。
- CF_SCREENFONTS——字体列表中只显示屏幕字体。

调用 ChooseFont 函数时，函数会停留直到对话框关闭为止，当用户单击了对话框中的“确定”按钮时，函数返回 TRUE，用户单击“取消”按钮退出时，函数返回 FALSE。具体的使用例子可以参考例子中的 _ChooseFont 子程序。

在调用 ChooseFont 之前，lpLogFont 字段被指向一个 LOGFONT 结构，对话框关闭的时候，函数在 LOGFONT 结构的 lfFaceName 字段中返回字体的名称，字体的效果和字形也在 LOGFONT 结构中返回。

用户选择的颜色在 rgbColors 字段中返回，字体大小在 iPointSize 字段中返回，由于单位是 1/10 磅，所以返回的数值等于对话框中选择的字体大小乘以 10。

8.2.3 “颜色选择”对话框

“颜色选择”对话框如图 8.3 所示，左边是基本的选择系统预定义颜色的区域，右边是扩展的区域，可以由用户自己选择或输入颜色值。

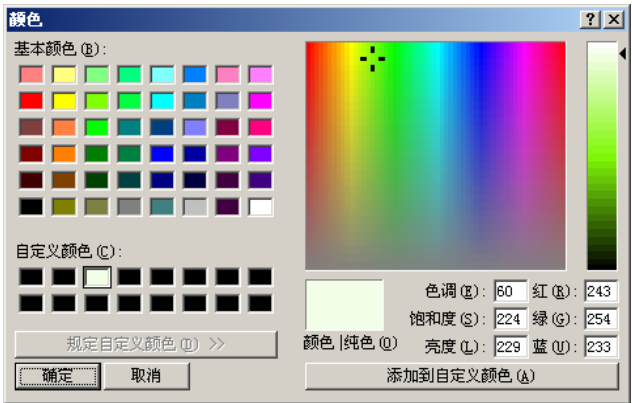


图 8.3 “颜色选择”对话框

打开“颜色选择”对话框使用函数 ChooseColor:

invoke ChooseColor, lpcc

lpcc 指向一个 CHOOSECOLOR 结构，这个结构是这样定义的:

CHOOSECOLOR		STRUCT		
LStructSize	DWORD	?	;结构长度	
HwndOwner	DWORD	?	;所属窗口	

HInstance	DWORD	?	
rgbResult	DWORD	?	;用户选择的颜色值
lpCustColors	DWORD	?	;用户自定义颜色缓冲区
Flags	DWORD	?	;标志
lCustData	DWORD	?	
lpfnHook	DWORD	?	
lpTemplateName	DWORD	?	
CHOOSECOLOR	ENDS		

结构中几个重要的字段说明如下。

- rgbResult——如果 Flags 字段指定了 CC_RGBINIT 标志,那么创建对话框的时候使用这个字段来初始化选择框中的颜色;函数返回的时候在这里返回用户选择的颜色。
- lpCustColors——指向一个 16 个双字长度的缓冲区,定义 16 种自定义颜色。
- Flags——标志,可以是下面取值的组合:
 - CC_FULLOPEN——对话框显示右边的扩展部分,如果不指定这个标志,初始化的时候扩展部分不显示,但用户可以通过单击“规定自定义颜色”按钮将对话框展开。
 - CC_PREVENTFULLOPEN——禁止“规定自定义颜色”按钮,也就是说不允许用户展开对话框的扩展部分。
 - CC_RGBINIT——对话框显示的时候用 rgbResult 字段的值初始化选择框中的颜色。

如果用户单击“确定”按钮,函数返回 TRUE,否则函数返回 FALSE。读者可以在例子文件中的_ChoseColor 子程序 z 序中找到使用这个函数的详细代码。

使用 ChooseColor 函数要注意的是:lpCustColors 指针不能为 NULL,所以必须分配一个 16 个双字长度缓冲区,如果指针是 NULL 会导致函数违规访问 00000000h 处的内存,就等着看“非法操作”吧!

8.2.4 “查找”和“替换”文本对话框

“查找”文本对话框如图 8.4 的下图所示,“替换”文本对话框如图 8.4 的上图所示,要显示这两种通用对话框可分别使用 FindText 和 ReplaceText 函数:

invoke	FindText, lpfr
invoke	ReplaceText, lpfr

这两个函数都使用同样的 FINDREPLACE 结构:

FINDREPLACEA	STRUCT		
lStructSize	DWORD	?	;结构长度
hwndOwner	DWORD	?	;所属窗口
hInstance	DWORD	?	
Flags	DWORD	?	;标志
LpstrFindWhat	DWORD	?	;查找字符串
lpstrReplaceWith	DWORD	?	;替换字符串
wFindWhatLen	WORD	?	;查找字符串长度
wReplaceWithLen	WORD	?	;替换字符串长度
lCustData	DWORD	?	

lpfnHook	DWORD	?
lpTemplateName	DWORD	?
FINDREPLACEA	ENDS	
FINDREPLACE	equ	<FINDREPLACEA>

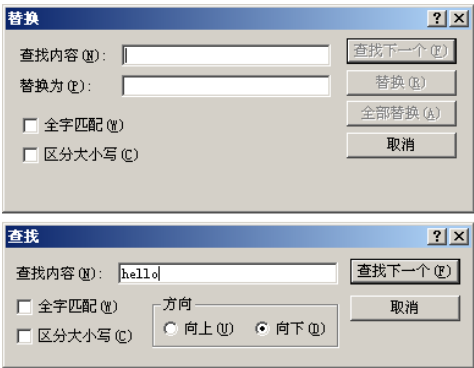


图 8.4 “查找”和“替换”对话框

结构中关键的字段说明如下。

- Flags——标志，创建对话框的时候，函数根据标志中的数值初始化对话框中各控件的状态，返回的时候根据用户的选择来设置标志字段的内容，标志字段可以是以下取值的组合：
 - FR_FINDNEXT, FR_REPLACE, FR_REPLACEALL, FR_DIALOGTERM——分别表示用户单击了“查找下一个”、“替换”、“全部替换”和“取消”按钮。
 - FR_HIDEUPDOWN, FR_HIDEMATCHCASE 与 FR_HIDEWHOLEWORD——初始化时使用，表示对话框中不显示“方向”、“区分大小写”与“全字匹配”按钮。
 - FR_NOMATCHCASE, FR_NOUPDOWN 与 FR_NOWHOLEWORD——初始化时将“区分大小写”、“方向”与“全字匹配”按钮灰化。
 - FR_MATCHCASE 或 FR_WHOLEWORD——表示用户选中了“区分大小写”或“全字匹配”复选框。
 - FR_DOWN——把“方向”单选钮设置为“向下”。
- lpstrFindWhat——指向包含查找字符串的指针，缓冲区的长度必须至少为 80 字节，这个字符串在初始化的时候出现在“查找内容”编辑框中，函数也在这里返回用户输入的内容。
- lpstrReplaceWith——指向包含替换字符串的指针，这个字符串在初始化的时候出现在“替换为”编辑框中，函数也在这里返回用户输入的内容。这个字段在使用 FindText 函数的时候可以为 NULL，但在使用 ReplaceText 函数的时候必须设置，否则对话框不会显示。
- wFindWhatLen 和 wReplaceWithLen——lpstrFindWhat 和 lpstrReplaceWith 指示缓存区的长度。

“查找”和“替换”对话框的使用有些特殊，因为这两种对话框是非模态对话框，也就是说，FindText 和 ReplaceText 函数被调用后，系统显示对话框后马上返回，对话框保持显示状态，直到用户按下了“取消”按钮后对话框才关闭。如果用户按下了对话框中的某个按钮，对话框设置 FINDREPLACE 结构的相关字段并通过自定义的消息通知父窗口的窗口过程，程序中处理查找和替换的功能集中在这个自定义消息中完成。另外，由于对话框必须向父窗口发送消息，所以 hwndOwner 字段中必须指定父窗口的句柄，而不能像其他通用对话框一样可以把 hwndOwner 字段设置为 NULL。

为了让对话框能够使用自定义消息，程序必须首先使用 RegisterWindowMessage 函数注册自定义消息，这个函数注册消息并返回消息 ID，输入的参数是消息名称字符串，Microsoft 的编程手册中说明要为查找和替换对话框注册 FINDMSGSTRING 消息，但没有任何资料说明 FINDMSGSTRING 究竟代表什么，是代表消息名称字符串为“FINDMSGSTRING”吗？不是，实际上它代表字符串“commdlg_FindReplace”，所以，在对话框的初始化消息中如下注册消息：

```
idFindMessage dd ?
FINDMSGSTRING db 'commdlg_FindReplace',0
...
invoke RegisterWindowMessage, addr FINDMSGSTRING
mov idFindMessage, eax
```

注意：要把 RegisterWindowMessage 函数返回的消息 ID 保存下来，并在主窗口的消息循环中通过判断 uMsg 是否等于这个消息 ID 来判断对话框是否发回消息：

```
mov     eax, uMsg
.if     eax == WM_XXX
...
.elseif eax == idFindMessage
.if     stFind.Flags & FR_DIALOGTERM
;用户按下了“取消”按钮, 对话框关闭
.elseif stFind.Flags & FR_FINDNEXT
;用户按下了“查找下一个”按钮
.elseif stFind.Flags & FR_REPLACE
;用户按下了“替换”按钮
.elseif stFind.Flags & FR_REPLACEALL
;用户按下了“全部替换”按钮
.endif
```

由于用户按下了“查找下一个”、“替换”、“全部替换”和“取消”等按钮时，对话框都要发回消息，所以在处理时还要根据不同的标志进行不同的处理。



因为查找和替换对话框是非模态对话框，所以使用时要把 FINDREPLACE 结构和字符串变量放在全局变量中，如果放在窗口过程的局部变量中，对话框还没有关闭的情况下，这些局部变量就已经被释放，以后对话框存取的就会是无效的地址。

8.2.5 “页面设置”对话框

“页面设置”对话框用来设置打印机参数，用户可以在对话框中选择打印机、打印纸张大小、页边距和纸张方向，还可以引出一个打印机属性的设置对话框。用户可以在这里完成与打印有关的所有设置工作，页面设置对话框如图 8.5 所示。

显示“页面设置”对话框使用 PageSetupDlg 函数：

```
invoke    PageSetupDlg, lppsd
```

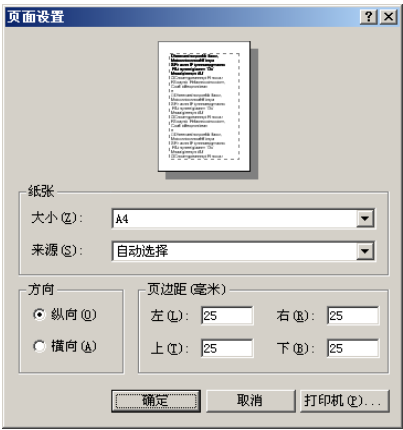


图 8.5 “页面设置”对话框

lppsd 参数指向一个 PAGESETUPDLG 结构：

PAGESETUPDLG	STRUCT			
LStructSize	DWORD	?		;结构长度
HwndOwner	DWORD	?		;所属窗口
hDevMode	DWORD	?		;指向 DEVMODE 结构
hDevNames	DWORD	?		;指向 DEVNAMES 结构
Flags	DWORD	?		;标志
PtPaperSize	POINT	<>		;返回纸张尺寸
RtMinMargin	RECT	<>		;返回最小允许的页边距
rtMargin	RECT	<>		;返回用户选择的页边距
hInstance	DWORD	?		
lCustData	DWORD	?		
lpfnPageSetupHook	DWORD	?		
lpfnPagePaintHook	DWORD	?		
lpPageSetupTemplateName	DWORD	?		
hPageSetupTemplate	DWORD	?		
PAGESETUPDLG	ENDS			

结构中的有关字段说明如下。

- hDevMode——如果用户选择了打印机，那么这里返回一个指针，指向包含 DEVMODE 结构的内存块地址，DEVMODE 结构中包含了打印机的名称。
- hDevNames——如果用户选择了打印机，那么这里返回一个指针，指向包含 DEVNAMES 结构的内存块的地址，DEVNAMES 结构包含了打印机的各种属性。

- Flags——标志，可以是以下取值的组合：
 - PSD_DEFAULTMINMARGINS——将页边距设置为打印机允许的最小页边距。
 - PSD_DISABLEMARGINS, PSD_DISABLEORIENTATION 和 PSD_DISABLEPAGEPAINTING——灰化页边距设置输入框、纸张方向选择框和纸张选择框。
 - PSD_DISABLEPAGEPAINTING——不绘画最上方的打印示例图示。
 - PSD_DISABLEPRINTER——灰化“打印机”按钮。
 - PSD_MARGINS——函数用 rtMargin 字段的值初始化对话框中的数值。
 - PSD_RETURNDEFAULT——函数不显示对话框，马上返回并在 hDevNames 和 hDevMode 字段中返回默认打印机的设置情况。
 - PSD_INTHOUSANDTHSOFINCHES 和 PSD_INHUNDREDTHSOFMILLIMETERS——指明 ptPaperSize, rtMinMargin 与 rtMargin 等字段使用的单位是英寸还是毫米。
- ptPaperSize——一个 POINT 结构，返回纸张大小。
- rtMinMargin——打印机允许的最小页边距。
- rtMargin——用户选择的页边距数据。

当用户选择了打印机时，hDevMode 中返回的是一个指向内存块的指针，所以需要用下面的代码获取 DEVMODE 结构的地址：

```
mov  eax, @stPS.hDevMode
mov  eax, [eax] ;现在 eax 是 DEVMODE 结构的地址
```



总结通用对话框的使用方法可以发现，每种通用对话框函数都使用一个特定的结构来当做输入输出的缓冲区，初始化的时候函数根据结构中的数据和标志设置对话框中的子窗口控件，返回的时候在结构的相应位置返回用户的输入或选择。另外，所有结构都有几个类似的字段，如 lStructSize 字段必须设置为正确的结构长度；hwndOwner 指定对话框的父窗口，模态对话框在关闭之前是不允许切换到这个父窗口中去的。

8.2.6 “浏览目录”对话框

在众多的系统提供的对话框中，还有一个很常用的浏览目录对话框，对话框如图 8.6 所示，这个对话框虽然也是通用型的，但是它是由 Shell32.dll 提供的，而不是由 Comdlg32.dll 提供的，在实现的方法上也和上面介绍的通用对话框有很大的不同，由于篇幅较大，有关该对话框的详细介绍放在附录 C 中（以电子版方式放在附书光盘中），有兴趣的读者可以自行阅读。



图 8.6 “浏览目录”对话框

控件是封装成“黑匣子”形式的具有特定功能的子窗口，可用来增强用户界面。使用控件不但可以让代码模块化，简化主程序的复杂性，还可以使用户界面标准化，第 5 章中介绍的子窗口控件就是一些好例子。

在对话框中使用的子窗口控件属于简单的控件，由于这些控件使用频繁且规模比较小，所以平时都被装入了内存中。但系统中也存在一些比较复杂的可以选用的控件，如工具栏、状态栏、列表视图控件、树型视图控件和日历等，这些控件比较复杂，所以将它们取出来放在几个单独的动态链接库中来实现，一般把它们称为通用控件（Common Controls），包含通用控件的库文件 Comctl32.dll 就是通用控件库（Common Controls Library）。

控件的概念应用的很广泛，比如 Visual C++ 中就有很多控件，但是它们是在 C 语言级别上提供的，已经经过了 Visual C++ 的封装，在汇编中无法使用。Win32 汇编中可以使用 Windows 在操作系统级别提供的控件，本章介绍的就是这一类控件，但如果要把这些控件全部详细介绍的话，内容会远远超出本书的篇幅，所以在本章中有选择地详细介绍工具栏、状态栏和 Richedit 等最常用和最有代表性的控件，使读者能够举一反三地根据资料了解其他控件的用法。

9.1 通用控件简介

9.1.1 通用控件的分类

由于大部分的通用控件由 Comctl32.dll 模块提供，所以在使用之前要在源程序中包含相应的 include 和 includelib 语句：

include	Comctl32.inc
includelib	Comctl32.lib

Comctl32.dll 中提供的通用控件如表 9.1 所示。

表 9.1 通用控件

控 件 名 称	预定义的窗口类	说 明	特 殊 风 格
Animation Controls	SysAnimate32	动画	ACS_
Header Controls	SysHeader32	标题栏	HDS_
ListView Controls	SysListView32	列表视图	LVS_
TreeView Controls	SysTreeView32	树型视图	TVS_
Tab Controls	SysTabControl32	项目列表	TCS_
Progress Bars	msctls_progress32	进度条	
Status Windows	msctls_statusbar32	状态栏	SBARS
HotKey Controls	msctls_hotkey32	热键	
Trackbars	msctls_trackbar32	跟踪条	TBS_
Up-Down Controls	msctls_updown32	滚动条	UDS_
Toolbars	ToolbarWindow32	工具栏	TBSTYLE_
Tooltip Controls	Tooltips_class32	提示文本	
ImageLists		图像列表	
PropertySheets		属性表格	
PropertySheetsPage		属性页	
DragList		能处理拖放功能的列表框	

在高版本的 Comctl32.dll (IE4.0 以上版本更新的 Comctl32.dll 文件) 中, 还包括了一些扩展的通用控件, 这些扩展控件如表 9.2 所示。

表 9.2 扩展通用控件

控 件 名 称	预定义的窗口类	说 明	特 殊 风 格
Rebar Controls	ReBarWindow32	IE 风格工具栏	RBS_
Date & Time Picker	SysDateTimePick32	日期时间	DTS_
IP Address Picker	SysIPAddress32	IP 地址输入	
Pager Controls	SysPager		PGS_
ComboBoxEx	ComboBoxEx32	扩展 ComboBox	CBS_

Windows 系统自身附带的软件中也大量使用通用控件, 以图 9.1 中所示的“资源管理器”程序界面为例, 窗口的上方使用标题栏控件, 标题栏控件上显示的说明文字是提示文本控件, 窗口下方使用状态栏控件; 左边列出目录的地方是树型视图控件, 右边列出文件的地方是列表视图控件, 列表视图控件中的标题栏本身就使用另一个控件——标题栏控件。

其他的一些控件在操作系统中也随处可见, 如跳格表控件通常在属性设置对话框中使用; 拷贝大文件时的进度窗口中有个进度条控件。

除了这些控件之外, Richedit 控件也是一个很常用的控件, Richedit 控件是 Edit 控件的增强版本, 包含了很完整的文本编辑功能, 可以用来编辑带格式的 rtf 文件和不带格式的 txt 文件, 由于该控件非常复杂, 代码的规模比较大, 单独一个 Richedit 控件的代码规模就和 Comctl32.dll 中全部代码的规模相当, 所以 Windows 系统将 Richedit 控件单独放到另一个 dll 文件中, 有关 Richedit 控件的情况将在 9.4 节中详细介绍。

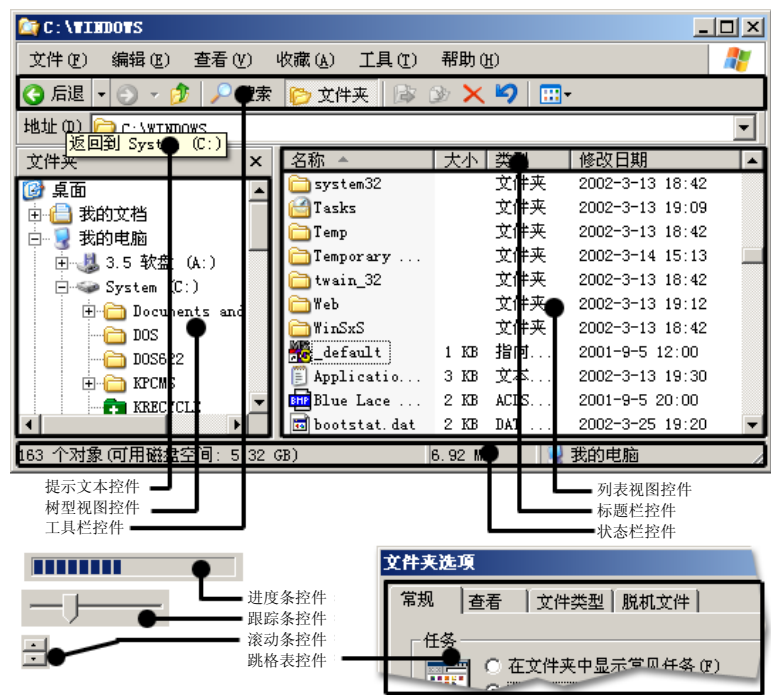


图 9.1 常见的通用控件

9.1.2 使用通用控件

1. 库初始化

通用控件的数量非常多，平时把它们全部装入并注册是非常浪费内存的，所以在默认状态下 Comctl32.dll 并不会被装入内存，因此，在使用通用控件之前必须将通用控件库装入内存，专用函数 InitCommonControls 可以用来完成这个工作，调用这个函数的唯一目的是保证系统加载 Comctl32.dll 库文件。

当库文件被装入的时候，库的入口函数会注册所有的通用控件类，然后用户程序就可以使用这些预定义的类来创建各种类型的通用控件窗口，这就像创建其他的子窗口控件一样。InitCommonControls 函数没有参数，也没有定义返回值，它的使用方法是：

invoke	InitCommonControls
--------	--------------------

InitCommonControls 函数仅注册表 9.1 中所列的通用控件类，并不注册表 9.2 中的扩展通用控件。如果需要使用扩展通用控件，那么需要使用 InitCommonControlsEx 函数来进行装入和注册的工作：

invoke	InitCommonControlsEx, lpInitCtrls
--------	-----------------------------------

lpInitCtrls 参数指向一个 INITCOMMONCONTROLSEX 结构：

INITCOMMONCONTROLSEX	STRUCT
dwSize	dd ?;结构长度

```
dwICC dd ?;需要初始化的类
INITCOMMONCONTROLSEX ENDS
```

结构中的 dwICC 字段指定了需要注册的扩展通用控件类，与 InitCommonControls 注册所有它支持的通用控件类不同，InitCommonControlsEx 函数只注册 dwICC 字段指明的扩展通用控件类，字段可以是下面取值的组合：

- ICC_BAR_CLASSES——注册工具栏、状态栏、Trackbar 和 Tooltip 类。
- ICC_COOL_CLASSES——注册 Rebar 类。
- ICC_DATE_CLASSES——注册 Date and Time Picker 类。
- ICC_HOTKEY_CLASS——注册 Hot Key 类。
- ICC_INTERNET_CLASSES——注册 IP Address Picker 类。
- ICC_LISTVIEW_CLASSES——注册 ListView 和 Header 类。
- ICC_PAGESCROLLER_CLASS——注册 Pager 类。
- ICC_PROGRESS_CLASS——注册 Progress Bar 类。
- ICC_TAB_CLASSES——注册 Tab 和 Tooltip 类。
- ICC_TREEVIEW_CLASSES——注册 TreeView 和 Tooltip 类。
- ICC_UPDOWN_CLASS——注册 Up-Down 类。
- ICC_USEREX_CLASSES——注册 ComboBoxEx 类。
- ICC_WIN95_CLASSES——注册 InitCommonControls 函数注册的所有类。

InitCommonControlsEx 函数是 InitCommonControls 函数的扩充，使用它也可以注册 InitCommonControls 函数能够注册的所有类（也可以仅注册其中的一部分），如果只用到通用控件，两个初始化函数都可以使用，但若用到扩展通用控件，那就只能使用 InitCommonControlsEx 函数来进行初始化了。

由于创建通用控件的代码一般放在主窗口的 WM_CREATE 消息中，所以 InitCommonControls 和 InitCommonControlsEx 函数的调用需要在此之前完成，一般在程序一开始的地方就调用它们。

2. 创建通用控件

因为大部分的通用控件都以窗口类的方法实现（惟一的例外是图像列表），所以创建通用控件窗口的方法和使用自定义窗口类建立窗口的方法是一样的，只要在 CreateWindowEx 函数中使用通用控件的类名就可以了。如果要在对话框中使用通用控件，也可以在资源文件中用定义子窗口控件同样的方法来定义通用控件（见 5.4.4 小节）。

在建立通用控件的时候，可以使用 WS_CHILD 等通用的窗口风格，除此之外，不同的通用控件也有自己的特殊风格，如树型视图控件有 TVS_XXXX 风格、列表控件有 LVS_XXXX 风格等，表 9.1 和表 9.2 中列出了一些特殊风格的前缀，这些风格的具体含义可以参考 Win32 API 函数指南。

可以在对话框资源定义中如下定义一个列表视图控件：

```
CONTROL "", IDC_LISTVIEW, "SysListView32",
    LVS_REPORT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
    11, 13, 216, 82, WS_EX_CLIENTEDGE
```

也可以在程序中使用 `CreateWindowEx` 函数如下创建：

```
szClass      db    "SysListView32", 0
...
invoke       CreateWindowEx, WS_EX_CLIENTEDGE, offset szClass, NULL, \
    LVS_REPORT or WS_CHILD or WS_VISIBLE or WS_BORDER, \
    11, 13, 216, 82, \
    hWinMain, IDC_LISTVIEW, hInstance, NULL
```

上述两种方法的使用效果是一样的，程序代码中的 `WS_CHILD` 等以 `WS` 开头的风格是窗口的通用风格，而 `LVS_REPORT` 风格是列表视图控件的特有风格。由于大多数时候通用控件都是当做子窗口创建的，所以窗口风格中必须包括 `WS_CHILD` 风格。另外，使用 `CreateWindowEx` 创建的时候必须指定 `WS_VISIBLE` 风格，否则控件不会被显示；而在对话框资源中定义的时候，系统总是默认加上 `WS_VISIBLE` 风格，所以有没有 `WS_VISIBLE` 是无所谓的。

除了调用 `CreateWindowEx` 函数来创建通用控件外，某些通用控件的创建可以使用一些专用的函数，这些函数其实在内部都调用了 `CreateWindowEx`，只是由于包装后的函数是量身定做的，使用起来更方便而已。经过包装的函数有：

- `CreateToolBarEx` 函数——用来创建工具栏。
- `CreateStatusWindow` 函数——用来创建状态栏。
- `CreateUpDownControl` 函数——用来创建滚动条控件。

另外，有些通用控件并没有自己的窗口类名称，或者说根本就不是窗口类，比如，属性表格和属性页控件是基于项目列表控件（Table Control）的，`DragList` 控件是基于下拉式列表框的，它们和其他控件的扩展；而图像列表控件本身是一幅图像而不是窗口类。创建这些控件的时候，由于无法用类名来指定它们，所以无法使用 `CreateWindowEx` 函数来创建，必须使用下面这些专用的创建函数：

- `PropertySheet` 函数——用来创建属性表格。
- `CreatePropertySheetPage` 函数——用来创建属性页。
- `ImageList_Create` 函数——用来创建图像列表。
- `MakeDragList` 函数——用来创建 `DragList` 控件。



在使用控件时要牢记的是：大部分的控件是窗口，它们是特殊的窗口，所以所有适用于窗口的概念都可以使用在这些以窗口为基础的控件上，包括创建与使用的方法、与父窗口的通信方式，以及内部的工作原理等。

3. 通用控件和父窗口之间的通信

当在对话框中使用子窗口控件时，父窗口通过 SendMessage 函数发送控制消息来管理子窗口控件，而子窗口控件通过发送 WM_COMMAND 或 WM_NOTIFY 消息来将用户的动作通知父窗口。

通用控件的通信方法和子窗口控件使用的方法是一样的——父窗口发送控制消息来管理通用控件，不同类型的通用控件使用不同的控制消息，如状态栏的控制消息都是以 SB_开头的（Status Bar 的缩写）；TreeView 控件的控制消息是以 TVM_开头的（Tree View Message 的缩写）；ListView 控件的控制消息是以 LVM_开头的（List View Message 的缩写）。不同的消息都有特定的用法和参数，在使用时需要查阅 Win32 函数手册。

通用控件也通过发送通知消息来与父窗口通信，不同通用控件使用的通知消息可能有所不同，归纳起来情况如下：

- 工具栏控件使用 WM_COMMAND 消息将按钮动作通知父窗口，这是为了便于和菜单、加速键使用同一份代码来处理用户按下工具栏按钮的动作。
- 滚动条控件使用 WM_VSCROLL 或者 WM_HSCROLL 消息通知父窗口，与窗口自身滚动条使用的消息名称保持一致可以便于使用已经存在的滚动条消息处理代码。
- 除了上面这两种特殊情况外，大部分通用控件使用 WM_NOTIFY 消息通知父窗口。这样可以避免和菜单或加速键等使用的 WM_COMMAND 消息相混淆。

当父窗口收到 WM_NOTIFY 消息时，wParam 参数的内容是通用控件的 ID，也就是使用 CreateWindowEx 函数创建控件时使用的第 10 个参数，通过这个参数可以判别 WM_NOTIFY 消息是由哪个通用控件发送的；消息的 lParam 参数指向一个 NMHDR 结构：

NMHDR STRUCT			
hWndFrom	DWORD ?	;发送 WM_NOTIFY 的通用控件的窗口句柄	
idFrom	DWORD ?	;发送 WM_NOTIFY 的通用控件的 ID	
code	DWORD ?	;通知码	
NMHDR ends			

通过 NMHDR 结构中的 hWndFrom 字段和 idFrom 字段也可以判别发送 WM_NOTIFY 消息的控件，由于使用 CreateWindowEx 函数创建多个通用控件的时候可以使用同样的 ID 值，所以有时候使用 ID 并不能惟一确定控件，只有在创建的时候对不同的控件使用了不同的 ID 值，才能用 ID 值来惟一确定控件。而系统中每个窗口的窗口句柄是惟一的，所以使用 hWndFrom 字段是肯定能惟一确定控件的。

结构中的 code 字段是通知码，通过这个字段可以了解到控件上发生的动作，每种控件都有自己独特的通知码集合，但下面的通知码是大部分控件都使用的：

- NM_CLICK——用户在控件上按下了鼠标左键。
- NM_DBLCLK——用户在控件上双击鼠标左键。
- NM_KILLFOCUS——控件失去了键盘输入焦点。
- NM_OUTOFMEMORY——控件在运行中内存耗尽。

- NM_RCLICK——用户在控件上按下了鼠标右键。
- NM_RDBLCLK——用户在控件上双击鼠标右键。
- NM_RETURN——用户在控件上按下了回车键。
- NM_SETFOCUS——控件得到了键盘输入焦点。

9.2 使用状态栏

状态栏一般位于主窗口的底部（当然，如果愿意的话，也可以把它放在主窗口的上方，不过几乎没有人这样做），用来显示程序运行中的一些状态信息。本节中的例子程序创建了一个带状态栏的对话框，运行界面如图 9.2 的左边窗口所示。状态栏中分别显示了时间、编辑器中的总字节数和插入状态等 3 栏内容，随着时间的改变和字符的输入，这些信息会随时被更新。状态栏的另一个重要应用是显示菜单项的说明信息。在例子程序中，随着鼠标移到不同的菜单项上，状态栏上的说明信息也随之改变，如图 9.2 的右边窗口所示。

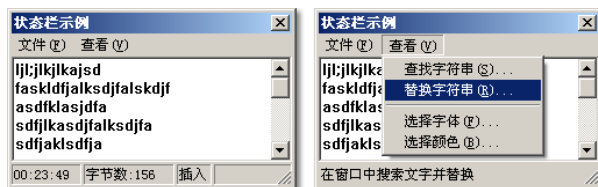


图 9.2 状态栏示例

一般来说，状态栏仅用于输出信息，并不用来输入信息，但有时也会使用状态栏来获取有限功能的输入，如在例子程序中状态栏的第3栏上单击鼠标，文字会在“插入”和“改写”之间切换，一些文本编辑软件就是用这种办法来改变文字输入方式的。

本节提供的例子位于所附光盘的 Chapter09\StatusBar 目录中，包括 StatusBar.asm 文件和 StatusBar.rc 文件。StatusBar.asm 文件的内容如下：

[illegible]

哭

```

        .if      eax ==    WM_TIMER
        invoke   GetLocalTime, addr @stST
        movzx    eax, @stST.wHour
        movzx    ebx, @stST.wMinute
        movzx    ecx, @stST.wSecond
        invoke   wsprintf, addr @szBuffer, addr szFormat0, \
                eax, ebx, ecx
        invoke   SendMessage, hWndStatus, SB_SETTEXT, \
                0, addr @szBuffer
;*****
        .elseif   eax ==    WM_CLOSE
        invoke   KillTimer, hWnd, 1
        invoke   EndDialog, hWnd, NULL
;*****
        .elseif   eax ==    WM_INITDIALOG
        mov      eax, hWnd
        mov      hWinMain, eax

        invoke   CreateStatusWindow, WS_CHILD OR WS_VISIBLE OR \
                SBARS_SIZEGRIP, NULL, hWinMain, ID_STATUSBAR
        mov      hWndStatus, eax
        invoke   SendMessage, hWndStatus, SB_SETPARTS, \
                4, offset dwStatusWidth
        mov      lpsz1, offset sz1
        mov      lpsz2, offset sz2
        invoke   SendMessage, hWndStatus, SB_SETTEXT, 2, lpsz1
        invoke   CreateWindowEx, WS_EX_CLIENTEDGE, \
                addr szClass, NULL, WS_CHILD or WS_VISIBLE or \
                ES_MULTILINE or ES_WANTRETURN or WS_VSCROLL or \
                ES_AUTOHSCROLL, \
                0, 0, 0, 0, hWnd, ID_EDIT, hInstance, NULL
        mov      hWinEdit, eax
        call     _Resize
        invoke   SetTimer, hWnd, 1, 300, NULL
;*****
        .elseif   eax ==    WM_COMMAND
        mov      eax, wParam
        .if      ax ==      IDM_EXIT
        invoke   EndDialog, hWnd, NULL
        .elseif  ax ==      ID_EDIT
        invoke   GetWindowTextLength, hWinEdit
        invoke   wsprintf, addr @szBuffer, \
                addr szFormat1, eax
        invoke   SendMessage, hWndStatus, SB_SETTEXT, \
                1, addr @szBuffer
        .endif
;*****
        .elseif   eax ==    WM_MENUSELECT
        invoke   MenuHelp, WM_MENUSELECT, wParam, lParam, \
                lParam, hInstance, hWndStatus, offset dwMenuHelp
        .elseif   eax ==    WM_SIZE
        call     _Resize
;*****
; 检测用户在第 3 栏的按鼠标动作并将文字在“插入”和“改写”之间切换

```

资源脚本文件 StatusBar.rc 的内容如下:

```
#include      <resource.h>
#define      ICO_MAIN      1000
#define      DLG_MAIN      1000
#define      IDM_MAIN      1000
#define      IDM_OPEN      1101
#define      IDM_SAVEAS      1102
#define      IDM_PAGESETUP  1103
#define      IDM_EXIT      1104
#define      IDM_FIND      1201
#define      IDM_REPLACE    1202
#define      IDM_SELFONT    1203
```

上述程序的结构和 8.1 节中演示通用对话框的例子几乎一模一样，使用的对话框和菜单都没有改变，但是源程序中将处理菜单项的代码全部去掉了，在菜单中保留这些菜单项仅为了演

示在状态栏上显示菜单提示信息的功能，菜单提示信息字符串被定义在资源文件的字符串表中。

程序在初始化对话框的 WM_INITDIALOG 消息（如果建立的是窗口而不是对话框，那么是 WM_CREATE 消息）中建立了一个状态栏和一个 EDIT 控件，并设置了一个定时器，用来在状态栏上显示时间。当窗口改变大小的时候，程序在 WM_SIZE 消息中重新安排状态栏和 EDIT 控件的位置。

另外，程序也处理状态栏发送的 WM_NOTIFY 通知消息，这是为了检测用户在第 3 栏上按下鼠标的动作，以便将文字在“插入”和“改写”之间切换，如果状态栏仅用于输出信息，那么就可以不处理 WM_NOTIFY 消息。

9.2.1 创建状态栏

创建状态栏可以使用 CreateStatusWindow 函数：

```
invoke    CreateStatusWindow, style, lpszText, hwndParent, wID
mov       hStatus, eax
```

style 参数指明状态栏的风格，它可以是以下取值的组合：

- SBARS_SIZEGRIP——显示状态栏右下角的斜条，用户可以拖动这里来改变主窗口的大小。
- CCS_TOP, CCS_BOTTOM 或 CCS_NOMOVEY——代表状态栏的初始位置，分别表示位于主窗口上方、下方（默认值）和禁止沿 Y 方向移动。
- CCS_NOPARENTALIGN——状态栏只自动设置自己的高度，不自动设置自己的宽度，也不自动移动位置。
- CCS_NORESIZE——禁止状态栏所有的自动移动和自动设置自己大小的特性，并禁止 CCS_TOP, CCS_BOTTOM, CCS_NOMOVEY 和 CCS_NOPARENTALIGN 风格。

lpszText 指向一个初始化的时候显示在状态栏上的字符串。hwndParent 指明状态栏的父窗口。wID 为状态栏控件的 ID，这个 ID 值可以用来在 WM_NOTIFY 消息中判断消息是否来自于状态栏。

在例子程序中，用以下代码建立了一个自动缩放的状态栏，状态栏的 ID 值被定义为 ID_STATUSBAR：

```
invoke    CreateStatusWindow, WS_CHILD OR WS_VISIBLE OR \
        SBARS_SIZEGRIP, NULL, hWinMain, ID_STATUSBAR
mov       hWinStatus, eax
```

当然，使用 CreateWindowEx 函数也可以完成同样的功能，只不过多了很多没有必要指定的参数而已：

```
szClass    db      'msctls_statusbar32', 0
...
invoke     CreateWindowEx, NULL, addr szClass, NULL, \
        WS_CHILD OR WS_VISIBLE OR SBARS_SIZEGRIP, \
```

```

                                0, 0, 0, 0, hWnd, ID_STATUSBAR, hInstance, NULL
mov     hWinStatus, eax

```

这几句语句的效果和使用 `CreateStatusWindow` 函数创建状态栏是一样的，语句中将位置和大小参数设置为 0 是因为状态栏有自动调整位置和大小能力。

9.2.2 状态栏的控制消息

1. 将状态栏分栏

状态栏刚建立的时候只有 1 栏，为了在状态栏上显示不同种类的信息，有时候需要将状态栏划分成多个栏目，如图 9.2 中的状态栏就被划分成了 4 栏，我们使用了前面的 3 栏来显示信息。

可以通过向状态栏发送 `SB_SETPARTS` 消息来将它分栏：

```

dwStatusWidth dd    60, 140, 172, -1
...
invoke    SendMessage, hWinStatus, SB_SETPARTS, 4, offset dwStatusWidth

```

其中 `SB_SETPARTS` 消息的 `wParam` 参数 4 代表所分的栏目数量，`lParam` 指向一个横向的坐标列表，坐标列表中的坐标数量和栏目数量相对应。如上面的代码将状态栏分为 4 栏，前面 3 栏分界点分别位于 X 方向的 60、140 和 172 处，最后一个坐标为 -1，表示最后一栏占据了剩下的所有宽度。如果最后一个坐标指定的不是 -1，比如说指定的是 60, 140, 172, 200，那么状态栏会是图 9.3 所示的样子——被分栏的地方是凹下的，剩余的地方则保持凸起的状态。



图 9.3 状态栏的分栏

在程序中也可以获取分栏的状态，通过向状态栏发送 `SB_GETPARTS` 消息就可以做到这一点：

```

dwStatusWidth dd    4 dup (?)
...
invoke    SendMessage, hWinStatus, SB_GETPARTS, 4, offset wStatusWidth

```

其中消息的 `wParam` 参数 4 代表用来接收分栏坐标的缓冲区的大小，`lParam` 参数指向用来接收分栏坐标数据的缓冲区，`SB_GETPARTS` 消息的返回值是状态栏的总栏数。

2. 维护状态栏中的信息

通过向状态栏发送 `SB_SETTEXT` 消息可以将字符串显示到指定的状态栏分栏中：

```

invoke    SendMessage, hWinStatus, SB_SETTEXT, iPart or uType, lpsz

```

消息的第一个参数指定分栏号和显示信息的方法，`iPart` 表示分栏号，分栏编号从 0 开始，`uType` 可以指定以下的方法：

- `SBT_NOBORDERS`——显示的文本不带边框（即分栏不显示为凹下的形状）。

- SBT_OWNERDRAW——分栏由用户自己绘画，当状态栏收到这样的消息后，会马上向父窗口发送 WM_DRAWITEM 消息，并在 WM_DRAWITEM 消息的 lParam 参数中指明一个 DRAWITEMSTRUCT 结构，这个结构中包括需要绘画的分栏的 hDC 与坐标等参数，程序可以用任何 GDI 函数对这个 hDC 进行绘画，包括使用 BitBlt 函数将一幅位图画到状态栏分栏中。
- SBT_POPOUT——默认状态下，状态栏的分栏显示为凹下，指定 SBT_POPOUT 标志将使分栏以凸起的形状显示。

消息的第二个参数 lParam 中放置的 lpsz 指向需要显示的字符串。

程序也可以通过发送 SB_GETTEXT 消息来获取状态栏中某个分栏的文字：

```
invoke    SendMessage, hWinStatus, SB_GETTEXT, iPart, lpsz
```

iPart 参数指定需要获取的分栏编号，lpsz 指向一个缓冲区，用来接收返回的字符串，由于消息的返回值是用 SB_SETTEXT 设置分栏文字时使用的 uType，所以返回值可能是 SBT_NOBORDERS 或 SBT_POPOUT 等数值。

在发送 SB_GETTEXT 消息之前，也可以首先通过发送 SB_GETTEXTLENGTH 消息来获取分栏中字符串的长度：

```
invoke    SendMessage, hWinStatus, SB_GETTEXTLENGTH, iPart, 0
and       eax, 0ffffh
mov       dwTextLength, eax
```

消息返回值的低 16 位是字符串的长度，高 16 位是用 SB_SETTEXT 设置分栏文字时使用的 uType。

3. 移动和缩放状态栏

当状态栏的父窗口改变大小的时候，程序必须移动和缩放状态栏以保证它以正确的尺寸位于正确的位置上。虽然以默认风格建立的状态栏是可以自动缩放和移动位置的，但这并不代表父窗口不用通知它。实际上，“自动缩放和移动位置”的含义是父窗口通知状态栏需要移动和缩放的时候，并不需要将正确的位置和大小告诉状态栏，新的位置和大小是由状态栏自己计算的。

所以，例子程序中当父窗口收到 WM_SIZE 消息的时候，用下面的代码来移动和缩放状态栏：

```
invoke    MoveWindow, hWinStatus, 0, 0, 0, 0, TRUE
```

我们看到，代码中并不需要指定有效的位置和尺寸。

如果创建状态栏的时候指定了 CCS_NOPARENTALIGN 或 CCS_NORESIZE 风格的话，那么在使用 MoveWindow 函数移动状态栏位置的时候就必须首先计算出正确的位置和大小。

9.2.3 在状态栏上显示菜单提示信息

本节演示当用户浏览菜单项的时候如何在状态栏上面显示菜单提示信息。每当用户将鼠标移过一个菜单项的时候，Windows 会向窗口过程发送一条 WM_MENUSELECT 消息，当用户真正选择了菜单项的时候，Windows 才会发送 WM_COMMAND 消息，为了显示菜单项提示信息，必须响应 WM_MENUSELECT 消息并根据浏览的菜单项显示对应的提示信息。

如果由我们自己实现这个功能，那么可以用 4 个步骤完成：首先是响应 WM_MENUSELECT 消息并从消息参数中获取用户浏览的菜单项 ID；第 2 步是根据菜单项 ID 获取对应的提示信息字符串；第 3 步是将状态栏的分栏取消以使用整个状态栏显示提示信息；第 4 步是在状态栏上显示字符串。

但在现实中不必这样做，在状态栏上面显示菜单提示信息的应用是这样的广泛，以至于系统为此设置了一个专用的函数 MenuHelp：

```
invoke    MenuHelp, uMsg, wParam, lParam, hMenu, hInstance, hwndStatus, lpwIDs
```

这个函数是和 WM_MENUSELECT 消息配合使用的，输入参数中的 uMsg, wParam 和 lParam 可以取自 WM_MENUSELECT 消息的对应参数，hMenu 是用户当前浏览的菜单句柄，由于 WM_MENUSELECT 消息的 lParam 参数传递过来的就是菜单句柄，所以函数的 hMenu 参数也可以直接使用 lParam。hwndStatus 参数指定了状态栏窗口的句柄。

hInstance 和 lpwIDs 配合指定了需要显示的提示信息字符串，MenuHelp 函数使用的字符串必须放在资源文件的字符串表中，函数会自动使用 LoadString 装入正确的字符串，参数 hInstance 指定了包含字符串资源的模块的实例句柄。lpwIDs 指向一个包含 4 个双字的数组，用来指定函数装入的字符串的 ID 值基数，第一个双字指定命令菜单项的基数，第二个双字指定弹出式菜单的基数。基数加上用户当前浏览的菜单项 ID 得到的数值就是 MenuHelp 函数要装入的字符串 ID。

举例说明，例子程序中调用 MenuHelp 函数如下：

```
dwMenuHelp      dd    0, IDM_MENUHELP, 0, 0
...
.elseif  eax ==  WM_MENUSELECT
    invoke  MenuHelp, WM_MENUSELECT, wParam, lParam, lParam, \
             hInstance, hWinStatus, offset dwMenuHelp
```

资源脚本文件中的字符串表定义如下：

```
stringtable      discardable
BEGIN
    IDM_MENUHELP      "包含文件操作的命令"
    IDM_MENUHELP+1    "包含操作视图的命令"

    IDM_OPEN           "打开需要编辑的文件"
    IDM_SAVEAS         "以另外一个文件名保存文件"
    IDM_PAGESETUP      "选择打印机以及设置页边距、纸张大小等打印参数"
    ...
```

由于 dwMenuHelp 数值第一项指定的基数为 0, 所以对于 IDM_OPEN 和 IDM_SAVEAS 等命令菜单项, MenuHelp 显示的字符串就等于菜单项 ID, 我们在资源脚本中只要将对应菜单项的提示字符串 ID 等同于菜单项 ID 定义就可以了。

dwMenuHelp 数值第二项指定的基数为 IDM_MENUHELP, 由于弹出式菜单没有 ID, 系统按照菜单的索引号加上基数当做字符串 ID, 所以对于第一个弹出式菜单“文件”, 函数显示的是 ID 号为 IDM_MENUHELP 的字符串, 对于第二个弹出式菜单“查看”, 菜单索引为 1, 所以函数显示的是 ID 为 IDM_MENUHELP+1 的字符串。

用例子程序中这种简单的逻辑可以显示命令菜单项和第一级弹出式菜单的提示信息, 但是无法显示第二级弹出式菜单的提示信息, 如果要显示第二级弹出式菜单的提示信息, 就必须根据 WM_MENUSELECT 消息的参数自己判断菜单是否是二级弹出式菜单, 并根据不同的情况调用不同参数的 MenuHelp 函数来实现。

9.3 使用工具栏

工具栏一般位于主窗口菜单栏的下方。工具栏也是一个子窗口, 它包含多个由位图组成的按钮, 工具栏上的按钮从功能上看和菜单项是类似的, 用户可以通过按动按钮来选择程序提供的各种功能。

工具栏上可以有不同种类的按钮, 有的按钮按下后会自动弹起, 有的按钮按下后保留在“选中”状态, 再按一次后恢复弹起状态, 按钮的“选中”状态可以是不互斥的或是互斥的, 另外, 按钮也可以被灰化或隐藏。所有这些按钮的属性和菜单项的属性是非常相似的, 所以工具栏往往用做菜单的补充, 为用户提供一个快捷的程序功能选择方式。

由于工具栏的主要用途是当菜单的补充, 为了与菜单逻辑使用同一套代码, 当用户按下工具栏上的按钮时, 工具栏向父窗口发送 WM_COMMAND 消息, 除了按动按钮的通知消息之外, 工具栏同样使用 WM_NOTIFY 消息将其他动作通知父窗口, 如用户拖动按钮来调整按钮的位置等。

工具栏上面的按钮看起来和对话框中的按钮很相似, 但实际上它们不是真正的按钮, 而仅是被工具栏控件绘画成按钮的样子罢了, 也就是说, 对话框中的按钮是子窗口, 而工具栏上的按钮并不是子窗口, 工具栏控件处理这些“仿真”按钮的方式就和一些图形界面的游戏一样, 在屏幕上绘画“模拟”的按钮样子并自行处理用户的鼠标动作, 以此检测用户在“按钮”上的动作。

本节的例子程序创建一个如图 9.4 所示的平面样式的工具栏, 当鼠标箭头移动到按钮上面的时候, 按钮会以凸起的形状显示, 鼠标停留片刻后, 出现一条简短的工具提示信息(图中鼠标箭头下方显示的“新建文件”)。

该例子程序代码在所附光盘的 Chapter09\Toolbar 目录中, 目录中包含汇编源文件 Toolbar.asm,

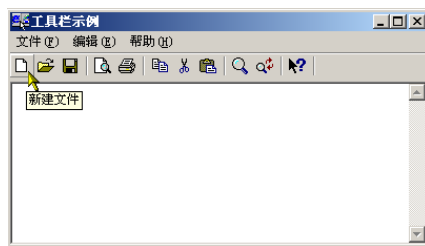


图 9.4 工具栏例子的运行结果

Toolbar.asm 文件的内容如下:

```
hInstance      dd      ?
hWinMain       dd      ?
hMenu          dd      ?
hWinToolbar    dd      ?
hWinEdit       dd      ?

;-----
; 工具栏
;-----
szClass        db      'EDIT',0
szClassName    db      'ToolbarExample',0
szCaptionMain  db      '工具栏示例',0
szCaption      db      '命令消息',0
szFormat       db      '收到 WM_COMMAND 消息, 命令 ID: %d',0
```

277

```

                                hWnd, ID_EDIT, hInstance, NULL
                                mov     hWinEdit, eax
                                invoke   CreateToolBarEx, hWinMain, WS_VISIBLE or \
                                WS_CHILD or TBSTYLE_FLAT or TBSTYLE_TOOLTIPS or \
                                CCS_ADJUSTABLE, ID_TOOLBAR, 0, HINST_COMMCTRL, \
                                IDB_STD_SMALL_COLOR, offset stToolbar, \
                                NUM_BUTTONS, 0, 0, 0, sizeof TBBUTTON
                                Mov     hWinToolbar, eax
                                Call _Resize
;*****
.elseif eax == WM_COMMAND
    mov     eax, wParam
    .if     ax == IDM_EXIT
        invoke DestroyWindow, hWinMain
        invoke PostQuitMessage, NULL
    .elseif ax != ID_EDIT
        invoke wsprintf, addr @szBuffer, \
        addr szFormat, wParam
        invoke MessageBox, hWnd, addr @szBuffer, \
        addr szCaption, \
        MB_OK or MB_ICONINFORMATION
    .endif
;*****
.elseif eax == WM_SIZE
    call _Resize
;*****
; 处理用户定制工具栏消息
;*****
.elseif eax == WM_NOTIFY
    mov     ebx, lParam
;*****
; 因为印刷宽度，请注意缩进格式！
;*****
.if [ebx + NMHDR.code] == TTN_NEEDTEXT
    assume ebx:ptr TOOLTIPTEXT
    mov     eax, [ebx].hdr.idFrom
    mov     [ebx].lpszText, eax
    push hInstance
    pop     [ebx].hinst
    assume ebx:nothing
.elseif ([ebx + NMHDR.code] == TBN_QUERYINSERT) || \
        ([ebx + NMHDR.code] == TBN_QUERYDELETE)
    mov     eax, TRUE
    ret
.elseif [ebx + NMHDR.code] == TBN_GETBUTTONINFO
    assume ebx:ptr TBNOTIFY
    mov     eax, [ebx].iItem
    .if     eax < NUM_BUTTONS
        mov     ecx, sizeof TBBUTTON
        mul     ecx
        add     eax, offset stToolbar
        invoke  RtlMoveMemory, addr [ebx].tbButton, \
        eax, sizeof TBBUTTON
        invoke  LoadString, hInstance, [ebx].tbButton.idCommand, \

```

哭

资源脚本文件 Toolbar.rc 的内容如下:

280

字符串表中定义的字符串是供工具提示信息使用的。

对话框过程的返回值是用来通知“对话框管理器”是否处理了相关消息的，这个返回值并不会被对话框管理器返回到工具栏子窗口去，对于大部分的控件来说，向父窗口发送 WM_NOTIFY 消息时并不需要父窗口回应一个返回值，但对于工具栏来说，父窗口必须根据 WM_NOTIFY 消息的处理情况返回 TRUE 或 FALSE，工具栏要根据返回值再做不同的动作，如果返回值无法返回，就意味着工具栏无法做正确的动作。

281

话框的按钮当做菜单快捷按钮的情况不在此列，因为这时只需要处理 WM_COMMAND 消息，WM_COMMAND 消息并不需要返回一个值。

例子程序中要演示用户定制工具栏按钮的功能，由于无法使用对话框当做主窗口，所以程序建立了一个常规的窗口。

9.3.1 创建工具栏

创建工具栏的专用函数是 CreateToolBarEx，使用 CreateWindowEx 函数利用类名“ToolbarWindow32”也可以创建工具栏，但 CreateWindowEx 函数仅创建一个空的工具栏，在创建完成后还要初始化工具栏，以及分多次插入按钮，而 CreateToolBarEx 函数可以一次创建工具栏，以及上面的全部按钮。

CreateToolBarEx 函数的用法是：

invoke	CreateToolBarEx, hwnd, ws, wID, nBitmaps, hBMInst, wBMID, lpButtons, \
	iNumButtons, dxButton, dyButton, dxBitmap, dyBitmap, uStructSize
mov	hToolbar, eax

各参数的说明如下。

hwnd 参数是父窗口的句柄，ws 是工具栏的风格，wID 是工具栏的子窗口 ID，这几个参数也可以在使用 CreateWindowEx 函数创建工具栏的调用中，ws 参数使用的窗口风格必须包括 WS_CHILD 和 WS_VISIBLE，另外还可以组合使用下面的特殊风格：

- TBSTYLE_FLAT——按钮的样式为平面样式，如果指定 TBSTYLE_FLAT 风格，创建的工具栏如图 9.5 上方的工具栏所示，如果不指定这个风格，则创建如图 9.5 下方所示的传统样式的工具栏。

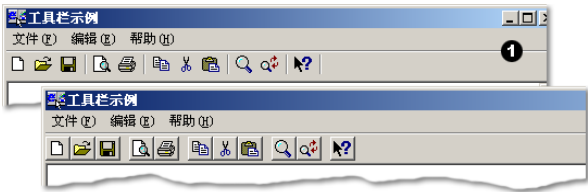


图 9.5 工具栏的风格

- CCS_NODIVIDER——工具栏边上没有分隔线，如果指定这个风格，那么图 9.5 中就不会有数字（1）标出的横线。
- TBSTYLE_WRAPABLE——工具栏支持多行显示。
- CCS_TOP, CCS_BOTTOM 或 CCS_NOMOVEY——指定工具栏的位置，分别表示工具栏位于窗口的上方（默认风格）、底部和不会自动在垂直方向移动。
- CCS_NOPARENTALIGN——工具栏自动设置自己的高度，但不自动设置宽度和位置。
- CCS_NORESIZE——禁止工具栏的自动缩放功能，相当于禁止了上面的 CCS_TOP、CCS_BOTTOM, CCS_NOMOVEY 和 CCS_NOPARENTALIGN 风格。

- CCS_ADJUSTABLE——允许用户在按下 Shift 键的同时通过拖动工具栏上的按钮来调整按钮位置以及删除按钮。
- TBSTYLE_ALTDRAW——对 CCS_ADJUSTABLE 风格的工具栏将拖动时的按键由 Shift 键改为 Alt 键。
- TBSTYLE_TOOLTIPS——工具栏支持工具提示信息。

hBMInst 和 wBMID 参数用来指定绘画工具栏上的按钮使用的位图，位图可以在资源中定义，也可以是已经装入内存的位图。如果使用资源中的位图，那么 hBMInst 指定包含位图资源的模块的实例句柄，wBMID 指定位图资源的 ID；如果 hBMInst 指定为 NULL，那就表示要使用一个已经装入内存的位图，这时 wBMID 必须指定一个合法的位图句柄。

指定了使用的位图以后，nBitmaps, dxBitmap 和 dyBitmap 参数继续给出了位图的属性，工具栏中的每个按钮并不使用一幅单独的位图，而是所有按钮的位图水平排列在一起组成一幅大的位图，nBitmaps 参数说明了这幅大位图中包含多少个按钮位图，dxBitmap 和 dyBitmap 参数指出了单个按钮位图的宽度和高度。显然，整个位图的高度就等于 dyBitmap，宽度等于 nBitmaps 乘以 dxBitmap。

图 9.6 是一幅典型的在工具栏中使用的位图，位图中包含了 15 个按钮。由于工具栏按照位置来分隔并使用位图，所以位图中各按钮的位置必须严格按照尺寸等距离排列，而且位图中不同按钮位图的尺寸必须是相同的。



图 9.6 工具栏使用的位图

除了使用自定义的位图以外，Comctl32.dll 库文件中也提供一些通用的位图供工具栏使用。为了使用这组位图，可以在 hBMInst 参数中使用预定义的模块实例句柄 HINST_COMMCTRL。

Comctl32.dll 包含了两组可以使用的位图。第一组就是如图 9.6 所示的位图，可以在 wBMID 中指定 IDB_STD_LARGE_COLOR (24×24 像素) 或 IDB_STD_SMALL_COLOR (16×16 像素) 来引用它们。这组位图包括 15 个按钮，Windows.inc 文件中已经为每个按钮的位置索引定义了预定义值，从左到右分别是 STD_CUT, STD_COPY, STD_PASTE, STD_UNDO, STD_REDO (不知道 REDO 后面为什么有个 W), STD_DELETE, STD_FILENEW, STD_FILEOPEN, STD_FILESAVE, STD_PRINTPRE, STD_PROPERTIES, STD_HELP, STD_FIND, STD_REPLACE 和 STD_PRINT。

第二组位图可以在 wBMID 中使用 IDB_VIEW_LARGE_COLOR 或 IDB_VIEW_SMALL_COLOR 来引用它们，这个位图包含 12 个按钮图像，索引值 0~11 分别被预定义为 VIEW_LARGEICONS, VIEW_SMALLICONS, VIEW_LIST, VIEW_DETAIL, VIEW_SORTNAME, VIEW_SORTSIZE, VIEW_SORTDATE, VIEW_SORTTYPE, VIEW_PARENTFOLDER, VIEW_NETCONNECT, VIEW_NETDISCONNECT 和 VIEW_NEWFOLDER，读者从字面上就可以想出这些位图究竟是什么样的。

参数 dxButton 和 dyButton 指定工具栏上按钮的尺寸，按钮的尺寸一般比按钮图像的尺寸要大一点。

剩余的 3 个参数 `lpButtons`, `iNumButtons` 和 `uStructSize` 用来定义工具栏上的按钮, 函数根据它们给出的数据创建工具栏上的全部按钮。 `lpButtons` 参数指向一组按顺序排列的 `TBBUTTON` 结构, 每个 `TBBUTTON` 结构定义一个按钮, `TBBUTTON` 结构的排列顺序决定了按钮在工具栏上的排列顺序; `iNumButtons` 参数指定工具栏上的按钮总数, 也就是 `lpButtons` 指向的数据中包含的 `TBBUTTON` 结构的总数; `uStructSize` 参数指明 `TBBUTTON` 结构的长度。

`TBBUTTON` 结构的定义如下:

<code>TBBUTTON</code>	STRUCT		
<code>iBitmap</code>	<code>DWORD</code>	?	;按钮使用的位图编号
<code>idCommand</code>	<code>DWORD</code>	?	;按钮按下时在 <code>WM_COMMAND</code> 中使用的 ID
<code>fsState</code>	<code>BYTE</code>	?	;按钮状态
<code>fsStyle</code>	<code>BYTE</code>	?	;按钮风格
<code>_wPad1</code>	<code>WORD</code>	?	;
<code>dwData</code>	<code>DWORD</code>	?	;自定义数据
<code>iString</code>	<code>DWORD</code>	?	;按钮字符串索引
<code>TBBUTTON</code>	<code>ENDS</code>		

上面的结构定义取自 MASM32 SDK 软件包所附带的 `Windows.inc` 文件, 但是 Microsoft Win32 API 手册中的结构定义并没有 `_wPad1` 字段, 现在并没有资料判定 `Windows.inc` 中的定义是否正确, 但这个定义在实际使用中并不会出错, 所以本书中的例子沿用这个定义, 这一点请读者注意。结构中各字段的含义如下。

- `iBitmap`——按钮使用的图像在 `wBMID` 参数指定的位图中的位置索引。位置索引从 0 开始, 也就是说第一个按钮图像的位置索引是 0。
- `idCommand`——按下按钮以后, 工具栏会向父窗口发送 `WM_COMMAND` 消息, 这个字段指定消息附带的命令 ID 号, 一般在这里指定与按钮对应的菜单项使用的 ID。
- `fsState`——按钮的类型和初始状态, 可以是下面取值的组合:
 - `TBSTATE_CHECKED`——按钮的类型是复选框按钮, 并且按钮初始化为选中状态 (即保持按下状态)。
 - `TBSTATE_ENABLED`——按钮被允许, 如果不指定这个标志, 按钮将显示为灰色, 并且不会接收用户的动作。
 - `TBSTATE_HIDDEN`——隐藏状态, 按钮不显示在工具栏上。
 - `TBSTATE_INDETERMINATE`——按钮处于灰化状态, 但可以接收用户的动作。
 - `TBSTATE_PRESSED`——按钮处于按下状态。
 - `TBSTATE_WRAP`——在包含 `TBSTYLE_WRAPABLE` 风格的多行工具栏中, 从此按钮开始换行。
- `fsStyle`——按钮风格, 可以是下面取值的组合:
 - `TBSTYLE_BUTTON`——标准按钮。
 - `TBSTYLE_CHECK`——复选框按钮 (按钮状态在按下和凸起之间切换)。

- TBSTYLE_GROUP——指定复选框按钮的分组边界。
- TBSTYLE_CHECKGROUP——TBSTYLE_CHECK 风格和 TBSTYLE_GROUP 风格的组合。
- TBSTYLE_SEP——按钮之间的分隔线。
- dwData——用户自定义数据。设置后可以通过 TB_GETBUTTON 消息查询。
- iString——按钮标记的索引。

在例子程序中，使用模块句柄 HINST_COMMCTRL 和位图句柄 IDB_STD_SMALL_COLOR 来指定使用 Comctl32.dll 中的预定义位图，并预定义了 16 个 TBBUTTON 结构，存放在常量 stToolbar 开始的地址中。

当使用预定义位图的时候，函数自己知道位图的大小和按钮的大小，所以位图和按钮的尺寸参数都可以设置为 0。函数的返回值是工具栏窗口的句柄，把它存放到 hWinToolbar 变量中以便在以后使用。

有关代码如下：

```

...
stToolbar    equ        this byte
TBBUTTON    <STD_FILENEW, IDM_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, -1>
TBBUTTON    <STD_FILEOPEN, IDM_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, -1>
TBBUTTON    <STD_FILESAVE, IDM_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, -1>
TBBUTTON    <0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, -1>
TBBUTTON    <STD_PRINTPRE, IDM_PAGESETUP, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, -1>

...
NUM_BUTTONS    EQU        16

...
invoke      CreateToolbarEx, hWinMain, WS_VISIBLE or WS_CHILD or TBSTYLE_FLAT \
            or TBSTYLE_TOOLTIPS or CCS_ADJUSTABLE, ID_TOOLBAR, 0, HINST_COMMCTRL, \
            IDB_STD_SMALL_COLOR, offset stToolbar, NUM_BUTTONS, \
            0, 0, 0, 0, sizeof TBBUTTON
mov         hWinToolbar, eax

```

9.3.2 工具栏的控制消息

程序可以通过向工具栏控件发送消息来控制工具栏，工具栏的控制消息比较多，下面分类进行讨论。

1. 工具栏的创建和维护消息

如果使用 CreateWindowEx 函数来创建工具栏，那么创建的是一个空白的工具栏，还需要对工具栏进行初始化。初始化的工作包括指定位图、指定 TBBUTTON 结构长度和添加按钮。

指定工具栏使用的位图使用 TB_ADDBITMAP 消息：

```

invoke      SendMessage, hToolbar, TB_ADDBITMAP, nButtons, lptbab

```

wParam 中的 nButtons 指定位图中包含的按钮图像数量，lptbab 指向一个 TBADDBITMAP 结构，结构中定义了两个字段：位图资源的模块句柄和位图 ID，这两个字段的定义方法与 CreateToolbarEx 函数的 hBMInst 和 wBMID 参数的定义方法是一样的。结构定义如下：

TBADDBITMAP STRUCT			
hInst	DWORD	?	;包含位图的模块实例句柄
nID	DWORD	?	;位图资源的 ID
TBADDBITMAP ENDS			

指定了位图以后，需要发送 TB_SETBITMAPSIZE 和 TB_SETBUTTONSIZE 消息来指定按钮图像的大小和按钮的大小，这一步相当于指定 CreateToolBarEx 函数中使用的 dxButton,dyButton,dxBitmap 和 dyBitmap 参数。

宽度和高度参数分别由 lParam 参数的高 16 位和低 16 位指定：

invoke	SendMessage,hToolbar,TB_SETBITMAPSIZE,0,dwWidth + dwHeight shl 16
invoke	SendMessage,hToolbar,TB_SETBUTTONSIZE,0,dwWidth + dwHeight shl 16

CreateToolBarEx 函数的最后一个参数 uStructSize 是为了向系统通知 TBBUTTON 的结构长度，如果使用 CreateWindowEx 函数来创建工具栏，那么这一步必须通过发送 TB_BUTTONSTRUCTSIZE 消息来完成：

invoke	SendMessage,hToolbar,TB_BUTTONSTRUCTSIZE,sizeof TBBUTTON,0
--------	--

接下来可以使用 TB_ADDBUTTONS 消息来添加按钮，uNumButtons 参数指定要添加的按钮数量，lpButtons 参数指向一组 TBBUTTON 结构，结构的数量和 uNumButtons 参数相对应，在发送这个消息前必须先发送 TB_BUTTONSTRUCTSIZE 消息指定结构的长度：

invoke	SendMessage,hToolbar,TB_ADDBUTTONS,uNumButtons,lpButtons
--------	--

TB_ADDBUTTONS 消息总是在工具栏的最后添加按钮，使用 TB_INSERTBUTTON 消息可以在几个按钮的中间插入新的按钮，但一次只能插入一个按钮，消息的 wParam 参数指定插入位置，lParam 消息指向一个 TBBUTTON 结构：

invoke	SendMessage,hToolbar,TB_INSERTBUTTON,iButtons,lpButton
--------	--

当然，如果使用 CreateToolBarEx 函数创建工具栏，那么上面的步骤就全部由函数包办了，这就是使用专用函数的好处。

也可以通过发送 TB_DELETEBUTTON 消息来删除工具栏上的按钮，iButton 参数指定按钮的位置索引，第一个按钮用 0 表示：

invoke	SendMessage,hToolbar,TB_DELETEBUTTON,iButton,0
--------	--

除了这些消息，还有一些消息可以用来获取工具栏的当前状态，如表 9.3 所示。

表 9.3 获取工具栏状态的消息

消 息	wParam	lParam	说 明
TB_BUTTONCOUNT	0	0	返回工具栏上按钮的数量
TB_GETBITMAP	idButton	0	返回指定按钮的图像索引
TB_GETBUTTON	iButton	lpButton	返回指定按钮的 TBBUTTON 结构
TB_GETROWS	0	0	返回多行工具栏当前包含的行数

TB_GETITEMRECT	iButton	lpRect	在 lParam 指定的位置返回包含指定按钮的位置的 RECT 结构
----------------	---------	--------	------------------------------------

所有消息的参数中，iButton 指按钮的位置索引，idButton 指按钮的命令 ID 值。

2. 移动和缩放工具栏

用默认参数建立的工具栏能够自动移动和缩放大小，当主窗口的宽度变宽的时候，即使不对工具栏进行调整，工具栏的宽度还是会自动扩展到父窗口的宽度。但有个小缺陷就是工具栏自动变宽的时候，图 9.5 中数字（1）所示的分隔线却不会自动变长，结果工具栏的外观似乎不是很好看，因此在主窗口的 WM_SIZE 消息中还是需要对工具栏进行调整。不过调整的方法很简单，只要对工具栏发送 TB_AUTOSIZE 消息就可以了：

```
invoke    SendMessage, hToolbar, TB_AUTOSIZE, 0, 0
```

消息中不必指定位置和大小参数，工具栏会自动计算新的大小，消息发送以后分隔线也会被调整到正确的宽度，一切看起来就完美了。

3. 工具栏按钮的维护消息

工具栏按钮可以像菜单项一样有选中、允许和灰化等状态，在程序中使用一组 TB_ISBUTTONxxxx 类型的消息来检测按钮的状态：

```
invoke    SendMessage, hToolbar, TB_ISBUTTONCHECKED, idButton, 0 ;是否在选中状态
invoke    SendMessage, hToolbar, TB_ISBUTTONENABLED, idButton, 0 ;是否在允许状态
invoke    SendMessage, hToolbar, TB_ISBUTTONHIDDEN, idButton, 0 ;是否在隐藏状态
invoke    SendMessage, hToolbar, TB_ISBUTTONINDETERMINATE, idButton, 0;是否灰化
invoke    SendMessage, hToolbar, TB_ISBUTTONPRESSED, idButton, 0 ;是否在按下状态
```

对于上面这些消息，如果答案是肯定的，那么消息返回 TRUE，否则消息返回 FALSE。如果嫌每次调用只能检测一种状态显得比较麻烦，也可以发送 TB_GETSTATE 消息：

```
invoke    SendMessage, hToolbar, TB_GETSTATE, idButton, 0
```

函数会返回按钮所有状态的组合值（TBSTATE_INDETERMINATE，TBSTATE_CHECKED，TBSTATE_ENABLED，TBSTATE_HIDDEN 或 TBSTATE_PRESSED 等状态的组合）。在上面这些消息中，idButton 用来指定按钮对应的命令 ID。

设置按钮的状态也可以通过一组消息来完成：

```
invoke    SendMessage, hToolbar, TB_CHECKBUTTON, idButton, uState ;选中按钮
invoke    SendMessage, hToolbar, TB_ENABLEBUTTON, idButton, uState;允许按钮
invoke    SendMessage, hToolbar, TB_HIDEBUTTON, idButton, uState ;隐藏按钮
invoke    SendMessage, hToolbar, TB_PRESSBUTTON, idButton, uState ;按下按钮
```

对于这些消息，如果 uState 指定为 TRUE，那么按钮会分别被设置为选中、允许、隐藏和按下状态；如果 uState 指定为 FALSE，按钮会被设置为非选中、灰化、显示和凸起的状态。

同样，要一次性设置所有状态，可以发送 TB_SETSTATE 消息：

```
invoke    SendMessage, hToolbar, TB_SETSTATE, idButton, uState
```

uState 参数可以指定为 TBSTATE_INDETERMINATE，TBSTATE_CHECKED，TBSTATE_ENABLED，

TBSTATE_HIDDEN 或 TBSTATE_PRESSED 等按钮状态的组合值。

9.3.3 工具栏的通知消息

大部分通用控件向父窗口发送的通知消息是 WM_NOTIFY，为了便于和菜单消息使用同一段命令处理的逻辑代码，当按动工具栏按钮的时候，工具栏控件向父窗口发送的是 WM_COMMAND 消息，但其他情况下发送的通知消息仍然是 WM_NOTIFY 消息。

工具栏发送的 WM_NOTIFY 消息主要用于显示工具提示和定制工具栏。

1. 工具提示

当工具栏的风格包含 TBSTYLE_TOOLTIPS 的时候，CreateToolBarEx 函数自动创建一个工具提示控件（Tool Tip），并为工具栏上的每个按钮注册提示文本，当鼠标指针移动到按钮上并停留片刻的时候，工具提示信息会自动显示出来。

因为工具提示信息是工具提示控件通过包含 TTN_NEEDTEXT 通知码的 WM_NOTIFY 消息向父窗口索取的，所以这个 WM_NOTIFY 消息严格地说应该属于工具提示控件的通知消息而不是工具栏的通知消息，但由于这里的工具提示控件是 CreateToolBarEx 函数自动创建的，所以在本书中还是一起介绍。

在包含 TTN_NEEDTEXT 通知码的 WM_NOTIFY 消息中，lParam 指向一个 TOOLTIPTTEXT 结构——慢着！前面不是说 WM_NOTIFY 消息的 lParam 参数指向一个 NMHDR 吗？怎么又是 TOOLTIPTTEXT 结构呢？这是因为不同控件的通知消息都使用 WM_NOTIFY 消息，有些通知消息可能需要附带其他数据，这时仅使用一个 NMHDR 结构来表达是不够的，Windows 的处理办法是为需要附带其他数据的 WM_NOTIFY 消息定义不同的数据结构，但这些结构头部都是一个 NMHDR 结构，NMHDR 结构以后才是其他字段，这样在得知通知码之前，把 lParam 参数指针当做一个 NMHDR 结构来处理总是正确的。而且只有先把 lParam 参数指针当做 NMHDR 结构处理并从中获取通知码以后，才真正知道 lParam 指向的究竟是什么结构。

好了，问题解决了，言归正传。TTN_NEEDTEXT 通知码的 lParam 指向一个 TOOLTIPTTEXT 结构，这个结构的定义是：

TOOLTIPTTEXT	STRUCT		
hdr	NMHDR	<>	;头部位置是一个 NMHDR 结构
lpszText	DWORD	?	;工具提示字符串指针
szText	BYTE 80 dup (?)		;工具提示字符串缓冲区
hInst	DWORD	?	;包含字符串资源的模块句柄
uFlags	DWORD	?	;标志
TOOLTIPTTEXT	ENDS		

当需要显示工具提示信息的时候，工具提示控件向父窗口发送 TTN_NEEDTEXT 通知码，父窗口将需要显示的提示字符串放在 TOOLTIPTTEXT 结构中并返回以后，工具提示控件就会把它显示出来。设置 TOOLTIPTTEXT 结构的办法有 3 种，读者可以任选其一：

（1）字符串包含在资源中，这时可以将 hInst 字段设置为包含资源的模块句柄，并把 lpszText 字段设置为字符串 ID，其他字段保持为 NULL，工具提示会自己使用 LoadString 函数

装入字符串。

(2) 将字符串放在内存中，将内存指针放入 lpszText 字段中，其他字段保持 NULL。

(3) 将字符串拷入 szText 字段中，其他字段保持 NULL。

例子程序使用了第一种办法。由于 NMHDR 结构的 idFrom 字段已经返回了按钮的命令 ID，所以在资源脚本文件中将字符串的 ID 和命令 ID 一一对应定义，然后使用第一种方法是最方便的，代码如下：

```
.elseif    eax ==    WM_NOTIFY
            mov      ebx, lParam
            .if      [ebx + NMHDR.code] == TTN_NEEDTEXT
                assume ebx:ptr TOOLTIPTEXT
                mov    eax, [ebx].hdr.idFrom
                mov    [ebx].lpszText, eax
                push   hInstance
                pop    [ebx].hinst
                assume ebx:nothing
            ...
```

读者可以自己尝试一下其他的方法。

2. 定制工具栏

定制功能是工具栏中最令人兴奋的特征：当工具栏包含 CCS_ADJUSTABLE 风格的时候，用户可以通过按下 Shift 键并拖动工具栏上的按钮来移动按钮位置；如果将按钮拖出工具栏的边界，按钮会被删除；更重要的是，如果向工具栏发送 TB_CUSTOMIZE 消息或者在工具栏的空白处双击鼠标，会显示出一个如图 9.7 所示的“自定义工具栏”对话框，对话框右边的列表框中列出了当前显示在工具栏上的按钮，左边列表框列出了可以添加到工具栏上的按钮，用户可以将一个按钮随意在使用和不使用之间切换，并且可以通过拖动按钮的上下位置来决定按钮在工具栏上的位置。



图 9.7 自定义工具栏对话框

工具栏通过一系列的通知信息来与父窗口交互，共同维护“自定义工具栏”对话框。在这个对话框建立和关闭的时候，工具栏通过 TBN_BEGINADJUST 和 TBN_ENDADJUST 通知码来通知父窗口；每次按钮被调整的时候，发送的是 TBN_TOOLBARCHANGE 通知码；在按下对话框中的“帮助”按钮和“重置”按钮的时候，发送的是 TBN_CUSTHELP 和 TBN_RESET 通知码，对于这些通知码，父窗口可以不必响应，这并不会影响对话框的使用。

与对话框是否能够正常运行有关的通知码是 TBN_QUERYINSERT, TBN_QUERYDELETE 和 TBN_GETBUTTONINFO, 父窗口必须应答这些通知码。

当用户在指定位置插入一个按钮的时候, 工具栏发送 TBN_QUERYINSERT 通知码询问父窗口是否允许此操作, 这时 lParam 指向一个 TBNOTIFY 结构, 这个结构定义如下:

```
TBNOTIFY STRUCT
hdr          NMHDR      <> ;显然, 这里肯定是 NMHDR 结构
iItem        DWORD      ?   ;按钮的位置索引
tbButton      TBUTTON   <> ;包含按钮信息的 TBUTTON 结构
cchText       DWORD      ?   ;pszText 中字符串的长度
pszText       DWORD      ?   ;按钮的说明字符串
TBNOTIFY ENDS
```

如果程序允许在此按钮前面插入一个新按钮, 那么返回 TRUE, 否则返回 FALSE。另外, 当自定义对话框刚显示的时候, 父窗口也会收到这个通知码, 这时必须返回 TRUE, 否则对话框在屏幕上一闪就消失了。

当用户要删除一个按钮的时候, 工具栏发送 TBN_QUERYDELETE 通知码, 询问父窗口是否允许此操作, 这时 lParam 也指向一个 TBNOTIFY 结构, 用来说明将要删除的按钮, 如果程序允许此操作则返回 TRUE, 否则返回 FALSE。

```
.elseif ([ebx + NMHDR.code] == TBN_QUERYINSERT) || \
        ([ebx + NMHDR.code] == TBN_QUERYDELETE) ;现在 ebx = lParam
        mov     eax, TRUE
        ret
```

在例子程序中使用上面的代码来处理这两个通知码, 也就是说, 对于全部的情况均返回 TRUE, 表示允许用户随意进行移动按钮和删除按钮的操作。

TBN_GETBUTTONINFO 通知码的处理就比较复杂了, 当工具栏需要全部按钮的信息的时候, 会多次发送 TBN_GETBUTTONINFO 通知码, 在例子程序中是这样处理的:

```
.elseif [ebx + NMHDR.code] == TBN_GETBUTTONINFO ;现在 ebx = lParam
        assume ebx:ptr TBNOTIFY ;lParam 也是指向一个 TBNOTIFY 结构
        mov     eax, [ebx].iItem
        .if     eax < NUM_BUTTONS
            mov     ecx, sizeof TBUTTON
            mul     ecx
            add     eax, offset stToolbar
            invoke  RtlMoveMemory, addr [ebx].tbButton, eax, sizeof TBUTTON
            invoke  LoadString, hInstance, [ebx].tbButton.idCommand, \
                addr @szBuffer, sizeof @szBuffer
            lea     eax, @szBuffer
            mov     [ebx].pszText, eax
            invoke  lstrlen, addr @szBuffer
            mov     [ebx].cchText, eax
            assume ebx:nothing
            mov     eax, TRUE
            ret
        .endif
```

首先来分析为什么要这样处理 TBN_GETBUTTONINFO 通知码。

工具栏控件每次总是发送一组 TBN_GETBUTTONINFO 通知码，并且每次 TBNOTIFY 结构中的 iItem 字段递增，父窗口需要每次在结构中返回一个按钮的信息，如果还有剩余的按钮信息没有告诉工具栏（比如，在用按钮和可选按钮加起来总共有 15 个，现在返回了 10 个，那么还剩 5 个按钮信息没有告诉工具栏），则在消息的返回值中返回 TRUE，工具栏由此知道还有多余的按钮，于是马上将 iItem 字段加 1 再次发送 TBN_GETBUTTONINFO 通知码，如此循环直到某一次消息的返回值是 FALSE 为止。

为什么工具栏不知道需要获取的按钮的数量，而需要由父窗口来确定呢？这是因为工具栏只维护栏上现存的按钮，当工具栏上当前有 10 个按钮的时候，如果在一组 TBN_GETBUTTONINFO 通知码中返回了 15 个按钮，这 15 个按钮中包括了已经在使用的 10 个按钮和可以添加上去的另外 5 个按钮，那么工具栏就会将这 15 个按钮与栏上现存的所有按钮比较，并把现存的 10 个按钮放在“自定义工具栏”对话框的右边，将剩余的 5 个放在对话框的左边。

在例子中可用的按钮总共是 16 个，如果在初始化的时候只需要显示前面 10 个按钮，那么在使用 CreateToolBarEx 函数的时候可以只指定 10 个按钮。在这种情况下，当定制工具栏时在一组 TBN_GETBUTTONINFO 通知码中返回全部 16 个按钮的时候，多余的 6 个按钮就会出现在对话框的左边。

另外，TBNOTIFY 结构的 pszText 需要返回按钮的说明文字，否则对话框中左右两个列表框中只会显示按钮图像而没有说明文字。程序在这里使用与工具提示信息同样的文字，因为这些文字存放在资源中，所以例子代码从 TBNOTIFY 结构包含的 TBUTTON 结构中取出 idCommand 字段，使用 LoadString 函数从资源中读取以 idCommand 为 ID 的字符串并将其放入 pszText 所指的缓冲区中，最后使用 lstrlen 函数求出字符串的长度并放入 cchText 字段中，这样对话框的列表框中就可以显示出按钮的名称字符串了。

9.4 使用 Richedit 控件

Richedit 控件与 Edit 控件类似，可以用于文本的输入和编辑。但两者在功能上各有侧重点。

Edit 控件广泛使用于对话框中，用来供用户输入少量的文字，因此加快速度和减少资源的占用是最重要的，而各种高级编辑功能不是主要的，也正因此，Edit 控件在短小精悍的同时，也存在诸多限制，最主要的就是在单行模式下，能容纳的文本不能超过 32 KB，在多行模式下也不能超过 64 KB。

Richedit 控件则侧重于文字的高级编辑功能，控件能够容纳的文本长度可以支持操作系统中的最大文件尺寸，并内置了很多高级编辑器才具有的特征，如多级的撤销或重做，向前或向后搜索，支持 Unicode 编辑等，最重要的就是支持 RTF (Rich Text Format) 格式的带段落格式的文本编辑。由于实现这些功能的代码比较复杂，所以 Richedit 控件的规模比较大，以至于 Windows 将它划分出来以一个单独的 DLL 库文件方式提供。

到目前为止，Richedit 控件总共有 3 个版本，这些版本的功能有所不同，总的来说高版本包

括了低版本的所有功能，但在某些细节的实现上又有些不同，随着版本的升高，一些设置工作也随之增多，所以如果不需要某些特殊功能的话，使用最高的版本可能并不是最适合的。

1.0 版本的 Richedit 控件对应的库文件是 Riched32.dll，Windows 95 只提供 1.0 版本，文件名中的 32 是 32 位版本的意思（不过并没有一个 Riched16.dll）。从 Windows 98 开始，系统中多了一个 2.0 版本的 Richedit 控件，Windows 2000 开始则有了 3.0 版本。2.0 版本和 3.0 版本的库文件名都是 Riched20.dll，同时 Riched32.dll 文件仍然存在于系统中，不过 Riched20.dll 文件名中的 20 总是让人迷惑，很多人第一次使用 Richedit 控件的时候误认为 Riched32.dll 的版本要比 Riched20.dll 的版本高。

除了在功能上的不同外，不同版本 Richedit 控件的类名称也有所不同，表 9.4 列出了 3 个版本之间的一些区别。

表 9.4 不同版本 Richedit 控件之间的区别

	1.0 版本	2.0 版本	3.0 版本
DLL 库文件名	Riched32.dll	Riched20.dll	Riched20.dll
控件的类名	Richedit	Richedit20A Richedit20W	Richedit20A Richedit20W
拖放编辑	支持	支持	支持
流输入输出	支持	支持	支持
Unicode 编辑	不支持	支持	支持
非窗口操作	不支持	支持	支持
自动 URL 识别	不支持	支持	支持
加速键	不支持	支持	支持
分行符	CR+LF	CR	CR（可模拟 1.0 版）
撤销/重做	支持单级	支持多级	支持多级
文本搜索	向前搜索	向前/向后搜索	向前/向后搜索

表 9.4 中列出的仅是一些最重要的区别，很多细微的区别并没有列出来，比如，每个版本都可以为文本设定下划线，但 3.0 版比 2.0 版又增加了点、划、划一点、划一点一点等多种样式的下划线。

Richedit 控件的 2.0 版本和 3.0 版本使用的控件名和类名是相同的，有时候为了使用某些版本特有的功能，需要预先检测版本号，但由于 Microsoft 并没有提供一个官方的检测方法，所以必须利用一些版本之间的区别来进行检测（这种方法好像在检测不同的 CPU），比如，排版样式功能（TYPOGRAPHY）是 3.0 版本才支持的，设置排版样式选项使用 EM_SETTYPOGRAPHYOPTIONS 消息，如果排版样式被设置后能够再检测到，说明控件的版本肯定是 3.0 的，代码如下：

```
invoke    SendMessage, hwndRichEdit, EM_SETTYPOGRAPHYOPTIONS, \
          TO_SIMPLELINEBREAK, TO_SIMPLELINEBREAK
invoke    SendMessage, hwndRichEdit, EM_GETTYPOGRAPHYOPTIONS, 1, 1
.if      eax==0          ;说明设置消息没被处理, 版本是 2.0 版
mov      dwVersion, 2
```

另外，也可以通过检测操作系统来确定 Richedit 控件的版本，如 2.0 版本在 Windows 98 和 Windows NT 4.0 中使用，而 Windows 2000 使用的是 3.0 版本。

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include        windows.inc
include        user32.inc
includelib     user32.lib
include        kernel32.inc
includelib     kernel32.lib
include        comdlg32.inc
includelib     comdlg32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Equ 等值定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN      equ        1000
IDA_MAIN      equ        2000
IDM_MAIN      equ        2000
IDM_OPEN      equ        2101
IDM_SAVE      equ        2102
IDM_EXIT      equ        2103
IDM_UNDO      equ        2201
IDM_REDO      equ        2202
IDM_SELALL    equ        2203
IDM_COPY      equ        2204
IDM_CUT       equ        2205
IDM_PASTE     equ        2206
IDM_FIND      equ        2207
IDM_FINDPREV  equ        2208
IDM_FINDNEXT  equ        2209
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .data?
hInstance     dd        ?
hWinMain      dd        ?
hMenu         dd        ?
hWinEdit      dd        ?
hFile         dd        ?
hFindDialog   dd        ?

```

294

哭

296

哭

```

        invoke    EnableMenuItem, hMenu, IDM_SAVE, MF_GRAYED
    .endif
;*****
    .if          szFindText
        invoke    EnableMenuItem, hMenu, IDM_FINDNEXT, MF_ENABLED
        invoke    EnableMenuItem, hMenu, IDM_FINDPREV, MF_ENABLED
    .else
        invoke    EnableMenuItem, hMenu, IDM_FINDNEXT, MF_GRAYED
        invoke    EnableMenuItem, hMenu, IDM_FINDPREV, MF_GRAYED
    .endif
;*****
    ret

_SetStatus    endp
;>>>>>>>>>
_Init        proc
    local     @stCf:CHARFORMAT

;*****
; 注册“查找”对话框消息，初始化“查找”对话框的结构
;*****
    push hWinMain
    pop     stFind.hwndOwner
    invoke  RegisterWindowMessage, addr FINDMSGSTRING
    mov     idFindMessage, eax
;*****
; 建立输出文本窗口
;*****
    invoke  CreateWindowEx, WS_EX_CLIENTEDGE, \
        offset szClassEdit, NULL, WS_CHILD OR WS_VISIBLE \
        OR WS_VSCROLL OR WS_HSCROLL OR ES_MULTILINE \
        or ES_NOHIDESEL, 0, 0, 0, hWinMain, 0, hInstance, NULL
    mov     hWinEdit, eax
    invoke  SendMessage, hWinEdit, EM_SETTEXTMODE, TM_PLAINTEXT, 0
    invoke  RtlZeroMemory, addr @stCf, sizeof @stCf
    mov     @stCf.cbSize, sizeof @stCf
    mov     @stCf.yHeight, 9 * 20
    mov     @stCf.dwMask, CFM_FACE or CFM_SIZE or CFM_BOLD
    invoke  lstrcpy, addr @stCf.szFaceName, addr szFont
    invoke  SendMessage, hWinEdit, EM_SETCHARFORMAT, 0, addr @stCf
    invoke  SendMessage, hWinEdit, EM_EXLIMITTEXT, 0, -1
    ret

_Init        endp
;>>>>>>>>>
_Quit        proc

    invoke  _CheckModify
    .if     eax
        invoke  DestroyWindow, hWinMain
        invoke  PostQuitMessage, NULL
        .if     hFile
            invoke  CloseHandle, hFile
        .endif
    .endif

```

哭

300

程序的资源脚本文件 Richedit.rc 的内容如下:

```
#include <resource.h>
#define ICO_MAIN 1000
#define IDA_MAIN 2000
#define IDM_MAIN 2000
```

302

END

由于篇幅所限，例子中仅演示了最基本的功能，一些附加的功能并没有写进去，如没有“另存为”功能（对打开的文件编辑后可以存盘，但是对直接在 RichEdit 控件中输入的内容没有设计存盘功能），也没有提供选择字体和颜色的对话框等，读者如果有兴趣的话，可以分析所附光盘的 Chapter09\Wordpad 目录中的程序，这是一个完整得多的编辑器，界面中使用了工具栏和状态栏，但由于全部代码的长度有 900 多行，所以在书中并没有列出它的代码。

Richedit.asm 程序中使用了一些第 8 章中介绍的通用对话框：当选择“打开”文件菜单的时候，使用 GetOpenFileName 显示一个“打开”文件对话框来供用户选择文件；另外，在选择“查找”菜单的时候，使用 FindText 函数显示一个“查找文本”对话框，所以在 WM_CREATE 消息中程序预先使用 RegisterWindowMessage 函数为“查找文本”对话框注册 FINDMSGSTRING 消息。对于这些内容，读者可以参看第 8 章中的相关章节。

9.4.1 创建 Richedit 控件

1. 装入 Richedit 控件

由于 Richedit 控件存在于一个单独的 DLL 库文件中，所以在使用前也要显式地装入库文件，装入 Richedit 库文件并不使用 InitCommonControls 之类的专用函数，一般使用通用的 LoadLibrary 函数来装入它，LoadLibrary 函数的用法是：

```
invoke    LoadLibrary, addr szDllName
mov       hDllInstance, eax
```

函数返回装入 DLL 的模块实例句柄，当不再使用库文件的时候，需要使用 FreeLibrary 函数将库释放。当库被装入时，库中的初始化代码会注册 Richedit 控件的窗口类，这样就可以在程序中利用 Richedit 的类名来创建控件。例子程序的_WinMain 子程序中是这样进行库的装入和释放工作的：

```
szDllEdit db      .const
                'RichEd20.dll', 0
...
.code
invoke    LoadLibrary, offset szDllEdit
mov       @hRichEdit, eax
...
; 主程序代码——创建窗口、消息循环等
...
invoke    FreeLibrary, @hRichEdit
```

当使用不同版本的 Richedit 控件时，注意要装入的库文件名是不同的。由于例子中使用 2.0 版本或 3.0 版本，所以装入的是 RichEd20.dll 文件。

2. 创建 Richedit 控件

创建 Richedit 控件的工作一般在主窗口的 WM_CREATE 消息中完成，创建的办法是使用 CreateWindowEx 函数：

```

        .const
szClassEdit db 'RichEdit20A',0
        ...
        .code
        invoke CreateWindowEx,WS_EX_CLIENTEDGE,offset szClassEdit,NULL,\
            WS_CHILD OR WS_VISIBLE OR WS_VSCROLL OR WS_HSCROLL \
            OR ES_MULTILINE or ES_NOHIDESEL,\
            0,0,0,0,hWinMain,0,hInstance,NULL
        mov     hWinEdit,eax

```

注意类名使用上的区别：1.0 版使用“RichEdit”，2.0 和 3.0 版本的类名有两种，ANSI 版本使用的类名是“RichEdit20A”，Unicode 版本使用的类名是“RichEdit20W”，例子中使用的是 ANSI 版本。

Richedit 控件可以使用的风格有 3 组：标准的窗口风格、Edit 控件风格和 Richedit 控件特有的风格。

在 Edit 控件可以使用的风格中，可供 Richedit 控件使用的有：

- ES_MULTILINE——可以编辑多行文字。
- ES_AUTOHSCROLL 和 ES_AUTOVSCROLL——自动滚动。
- ES_NOHIDESEL——失去键盘输入焦点的时候仍然显示选择区域。
- ES_READONLY——只读属性。
- ES_CENTER, ES_RIGHT 和 ES_LEFT——文本的对齐方式。
- ES_WANTRETURN——允许用户按回车键插入新的行。

但下列 Edit 控件风格不能在 Richedit 控件中使用：

- ES_LOWERCASE 或 ES_UPPERCASE——将控件中的文字全部转换成小写或大写。
- ES_PASSWORD——将控件中的文字显示为密码方式（显示为星号）。

下面列出的是部分 Richedit 控件特有的风格：

- ES_DISABLENOSCROLL——指定这个风格后，控件在不需要滚动条的时候（文字没有超出客户区大小）显示灰化状态的滚动条，而默认情况下，当不需要滚动条的时候，控件根本不会显示它。
- ES_NOIME——禁止输入法（IME）操作，仅用于亚洲语言版本。
- ES_SAVESEL——在失去键盘输入焦点的时候保存当前选择区域，默认状态下当控件在重新获得焦点的时候会将全部文本选中。

创建控件以后，需要发送 EM_EXLIMITTEXT 消息设置控件中能够容纳字符的总数，虽然 Richedit 控件中的文字长度可以支持到最大的文件尺寸，但由于默认情况下，控件还是将最大字符数限制为 64 KB，所以如果读者发现 Richedit 控件也只能编辑 64 KB 字符的话，那并不是控件的错，而是因为你没有告诉它具体的要求，EM_EXLIMITTEXT 消息的使用方法是：

```

        invoke     SendMessage,hWinEdit,EM_EXLIMITTEXT,0,dwTextMax

```

其中 dwTextMax 指定了最大字符数。

9.4.2 Richedit 控件的控制消息

1. 选择区域

选择区域就是用户在控件中通过拖动鼠标来选定的多个文字，选择了一段文字以后，用户以后的操作就是针对这段文字的，如按下 Delete 键可以删除整段文字，按 Ctrl+C 键将这段文字拷贝到剪贴板中，按 Ctrl+V 键用剪贴板中的内容替换这段文字等。

在程序中，选择区域可以被程序获取，也可以由程序自由设置，从编程角度来看，选择区域的用处有两个：

- 选定操作文本——与用户手工选定一段文本以便进行各种操作类似，程序在对文本进行操作之前也需要预先设置选择区域。
- 定位光标——控件中并没有专门的用来定位光标的控制消息，定位光标也是靠设置选择区域完成的。如果把选择区域的起始位置和结束位置设置为相同的，那么就相当于把光标定位到这个位置而不选定任何文字。

在 Edit 控件中，获取选择区域可以通过向控件发送 EM_GETSEL 消息：

```
invoke    SendMessage, hRichedit, EM_GETSEL, lpdwStart, lpdwEnd
```

lpdwStart 和 lpdwEnd 指向两个用来返回选定区域起始位置和结束位置的双字变量，也可以将这两个参数全部设置为 NULL，因为消息的返回值也是位置数据，返回值的低 16 位是选定区域的起始位置，高 16 位是结束位置。

但是 EM_GETSEL 消息仅适用于控件中文本长度不超过 64 KB 的情况，如果 Richedit 中选择区域的起始位置或结束位置有一个落在了 64 KB 以外，那么消息仅返回 -1，而不是正确的数值，所以最好还是使用 EM_EXGETSEL 消息，EM_EXGETSEL 消息是 Richedit 的特有消息，不能在 Edit 控件中使用：

```
invoke    SendMessage, hRichedit, EM_EXGETSEL, 0, lpchr
```

lpchr 参数指向一个 CHARRANGE 结构，用来接收选择区域的起始和结束位置，该结构定义如下：

```
CHARRANGE STRUCT
    cpMin  DWORD    ?           ;选择区域的起始位置
    cpMax  DWORD    ?           ;选择区域的结束位置
CHARRANGE ENDS
```

如果 cpMin 字段等于 cpMax 字段，表示选择区域的长度为 0，而光标位于这个位置；如果 cpMin 等于 0 而 cpMax 等于 -1，表示选定的是控件中的所有内容。

程序也可以通过发送对应的消息来设置选择区域：

```
invoke    SendMessage, hRichedit, EM_SETSEL, dwStart, dwEnd
invoke    SendMessage, hRichedit, EM_EXSETSEL, 0, lpchr
```

EM_SETSEL 的参数中直接用 dwStart 和 dwEnd 指定选择区域的开始和结束位置，但是这个消息同样有 64 KB 长度的限制；EM_EXSETSEL 消息没有这个限制，lpchr 参数同样指向一个 CHARRANGE 结构，结构中包含需要设定的位置。如果仅是移动光标而不选择任何区域，可以将起始位置和结束位置设置为相同的数值。

在程序中设置了选择区域（或改变了光标位置）后，可能这个区域与原来的选择区域位置相差太多，以至于落在了客户区的外面，用户已经看不到它了，如果希望控件能够滚动文字以便将新的位置落在客户区中，可以发送 EM_SCROLLCARET 消息，这个消息没有任何参数：

```
invoke    SendMessage, hRichedit, EM_SCROLLCARET, 0, 0
```

在例子程序中，根据是否存在选择区域来决定是否允许拷贝和剪切功能。因为如果不存在选择区域，就没有文本可供拷贝或剪切，所以在_SetStatus 子程序中使用下面的代码首先获取选择区域，并根据情况允许或禁止拷贝和剪切菜单项：

```
invoke    SendMessage, hWinEdit, EM_EXGETSEL, 0, addr @stRange
mov       eax, @stRange.cpMin
.if       eax == @stRange.cpMax; 不存在选择区域
    invoke EnableMenuItem, hMenu, IDM_COPY, MF_GRAYED
    invoke EnableMenuItem, hMenu, IDM_CUT, MF_GRAYED
.else     ; 存在选择区域
    invoke EnableMenuItem, hMenu, IDM_COPY, MF_ENABLED
    invoke EnableMenuItem, hMenu, IDM_CUT, MF_ENABLED
.endif
```

另外，在查找文本的_FindText 子程序中，一开始也通过发送 EM_EXGETSEL 消息获取选择区域，这是为了获得光标位置以便设置查找的起始点，当找到文本以后，文本的位置在 FINDTEXT 结构的 chrgText 字段中返回，chrgText 字段本身是一个 CHARRANGE 结构，所以直接在 EM_EXSETSEL 消息中使用它就可以将选择区域设置到找到的文字上：

```
invoke    SendMessage, hWinEdit, EM_EXSETSEL, 0, addr @stFindText.chrgText
invoke    SendMessage, hWinEdit, EM_SCROLLCARET, NULL, NULL
```

最后程序发送 EM_SCROLLCARET 消息滚动文字，以便找到的文本能够出现在用户的视野中。

2. 文本管理

文本管理涉及获取文本、设置文本，以及一些辅助操作。

因为 Richedit 控件本身就是一个窗口，所以可以通过常规的函数对其中的文本进行操作，比如，要获取和设置文本，可以调用 GetWindowText 或 SetWindowText 函数，也可以通过发送 WM_GETTEXT 和 WM_SETTEXT 消息来完成；如果需要获取控件中的文本长度，可以通过 GetWindowTextLength 函数或发送 WM_GETTEXTLENGTH 消息，不过所有这些操作针对的都是控件中的全部文字，无法实现细微的操作。

向控件发送控制消息仅针对选择区域操作则灵活得多，当通过 EM_EXSETSEL 消息设置好选择区域后，再通过 EM_GETSELTEXT 消息就可以获取当前选定的文本：

```
invoke    SendMessage, hWinEdit, EM_GETSELTEXT, 0, lpBuffer
```

lpBuffer 用来指定接收文本的缓冲区, 由于没有参数指定缓冲区的大小, 所以程序必须使用足够大的缓冲区, 不过这不是问题, 因为通过检查选择区域可以预先得知返回文本的大小, 消息的返回值是返回到缓冲区中的字符串的长度 (不包括末尾的 0 字符)。

通过发送 EM_REPLACESEL 消息可以替换选择区域中的文本, 如果当前的选择区域长度不为 0 的话, 选择区域的文本被消息指定的字符串所代替, 如果选择区域长度为 0, 则指定的字符串被插入到当前光标位置:

invoke	SendMessage, hWndEdit, EM_REPLACESEL, fCanUndo, lpString
--------	--

其中 fCanUndo 参数指出本次替换操作是否可以撤销, 如果指定 TRUE, 则控件保存撤销信息以便用户可以按 Ctrl+Z 键进行撤销, 指定 FALSE 的话操作就不能被撤销。lpString 参数指向插入或替换用的字符串, 字符串以 0 结尾。

除了获取和设置文字, 还有一系列的控制消息可以用来进行定位操作, 比如, 想要选定一整行内容, 就必须知道某一行的起始位置和结束位置; 另外, 有时候也需要在字符位置和行号之间进行转换计算, 对于这些要求, 把所有文本从控件中读出来再自己进行处理显然是很麻烦的, 幸好 Richedit 控件已经提供了这些功能。

通过发送 EM_EXLINEFROMCHAR 消息可以得知指定的字符位于哪一行中:

invoke	SendMessage, hWndEdit, EM_EXLINEFROMCHAR, 0, dwCharPos
mov	dwLine, eax

其中 dwCharPos 指出字符的位置 (以 0 开始), 消息将返回字符所处的行号。在所有这些消息中, 字符位置和行号都是从 0 开始计算的, 也就是说第 1 行的行号用 0 表示。

EM_LINEINDEX 消息则完成逆运算, 它返回指定行号的第一个字符的位置, dwLine 参数为输入的行号, 如果 dwLine 参数输入 -1 的话, 代表的是当前行 (光标所在的行):

invoke	SendMessage, hWndEdit, EM_LINEINDEX, dwLine, 0
mov	dwCharPos, eax

该消息返回指定行的起始字符的位置, 如果指定的行号超过了控件中文本的总行数, 那么消息将返回 -1。控件中包含文本的总行数可以通过 EM_GETLINECOUNT 消息获取:

invoke	SendMessage, hWndEdit, EM_GETLINECOUNT, 0, 0
mov	dwTotalLines, eax

如果想获取某一行的长度, 有好几种方法, 比如, 可以两次使用 EM_LINEINDEX 消息获取本行和下一行文字的起始位置, 再相减就得出了行的长度; 也可以用 EM_LINELENGTH 直接获取:

invoke	SendMessage, hWndEdit, EM_LINELENGTH, ich, 0
mov	dwLineLength, eax

不过, 由于 EM_LINELENGTH 消息中的 ich 参数并不是行号, 而是行中任意一个字符的位置, 所以想以行号为参数获取行长度的话, 还需要先用 EM_LINEINDEX 消息将行号转换到字符位置后再使用 EM_LINELENGTH 消息。这个消息还有个特殊用途, 当 ich 参数指定为 -1 的时候, 返回值是选定区域跨越的多个行中没有被选定的字符的总数, 这有什么用处呢? 显然, 当按下了 Delete

键删除了选择区域时，剩下的行的长度就是这个返回值。

要想获取某一行的内容也有多种办法，比如，可以先选定某个行，再用 EM_GETSELTEXT 消息来完成，但最简单的办法是使用 EM_GETLINE 消息：

```
mov     word ptr szBuffer, sizeof szBuffer
invoke  SendMessage, hWinEdit, EM_GETLINE, dwLine, addr szBuffer
```

其中 dwLine 参数指定要获取的行号，szBuffer 是用来接收字符串的缓冲区，注意：缓冲区的第一个字（不是双字！）必须预先指定为缓存区的长度！另外，由于接收的字符串并不包括结束符 0，所以在发送消息之前最好先把缓冲区全部清零，否则会 and 缓冲区中原有的数据混在一起。消息的返回值是返回到缓冲区中的字符串的长度。

3. 设置文本格式

Richedit 控件支持两种模式：带格式文本 RTF (Rich Text Format) 模式和不带格式文本 (Plain Text) 模式。在默认状态下控件处于 RTF 模式，在这种模式下，程序可以对控件中的不同文字分别设置不同的格式，这些格式可以被保存到 *.rtf 文件中。而在 Plain Text 模式下，只能将控件中的全部文字设置统一的格式，而且这些格式仅表现在“显示”上，不会被保存到 *.txt 文件中。

在控件窗口被创建后可以通过发送 EM_SETTEXTMODE 消息来设置工作模式，这条消息仅对 2.0 版本以上的 Richedit 控件有效：

```
invoke  SendMessage, hWinEdit, EM_SETTEXTMODE, dwTextMode, 0
```

当 dwTextMode 参数指定为 TM_PLAINTEXT 的时候，控件切换到不带格式模式；指定为 TM_RICHTEXT 的时候，控件切换到 RTF 模式。这个消息也可以用来设置重做/撤销的模式，在 dwTextMode 参数中同时指定 TM_SINGLELEVELUNDO 标志可以将控件设置为单级重做/撤销模式；指定 TM_MULTILEVELUNDO 标志则设置为多级重做/撤销模式。

要设置文本格式可以通过发送 EM_SETCHARFORMAT 消息，这个消息设置控件中一段选定的文本或者全部正文的格式，消息的用法如下：

```
invoke  SendMessage, hWinEdit, EM_SETCHARFORMAT, uFlags, lpFmt
```

uFlags 参数表示指定的格式所应用的范围，它可以是下面的数值。

- SCF_ALL——为控件中的全部文本设置指定的格式。
- SCF_SELECTION——仅为选择区域设置指定的格式，如果选择区域为空，则以后在此位置插入的新字符使用此格式。
- SCF_WORD 与 SCF_SELECTION——将格式应用到选定的单词上，如果选择区域没有落在整个单词上，那么格式会扩展到整个单词上，SCF_WORD 标志必须和 SCF_SELECTION 标志一起使用。

lpFmt 参数则指向一个 CHARFORMAT 或 CHARFORMAT2 结构，CHARFORMAT 结构可以在所有版本中使用，而 CHARFORMAT2 结构仅可以在 2.0 及以上版本使用，CHARFORMAT2 结构是 CHARFORMAT

结构的扩展，下面是 CHARFORMAT2 结构的定义：

CHARFORMAT2	STRUCT		
cbSize	DWORD	?	;结构长度
dwMask	DWORD	?	;字段掩码
dwEffects	DWORD	?	;文字效果
yHeight	DWORD	?	;文字高度
yOffset	DWORD	?	
crTextColor	DWORD	?	;文本颜色
bCharSet	BYTE	?	
bPitchAndFamily	BYTE	?	
szFaceName	BYTE	LF_FACESIZE	dup(?) ;字体名称
;CHARFORMAT 结构的定义到此为止			
wWeight	WORD	?	
sSpacing	WORD	?	
crBackColor	DWORD	?	
lcid	DWORD	?	
dwReserved	DWORD	?	
sStyle	WORD	?	
wKerning	WORD	?	
bUnderlineType	BYTE	?	
bAnimation	BYTE	?	
bRevAuthor	BYTE	?	
bReserved1	BYTE	?	
CHARFORMAT2	ENDS		

CHARFORMAT2 结构中 szFaceName 字段以前的内容就是 CHARFORMAT 结构，结构中各字段的含义如下。

- cbSize——结构的大小，由于控件使用该字段来判断结构的版本是 CHARFORMAT 还是 CHARFORMAT2，所以在将结构传递给控件前必须将这个字段设置为正确的数值。
- dwMask——字段掩码，用来指定结构中哪些字段是有效的，如果没有使用对应的标志，即使某些字段的内容被设置，控件也不会使用它，dwMask 中可以使用的标志可以是下面数值的组合：
 - CFM_BOLD——dwEffects 字段中 CFE_BOLD 值是有效的。
 - CFM_CHARSET——bCharSet 字段是有效的。
 - CFM_COLOR——crTextColor 字段和 dwEffects 中的 CFE_AUTOCOLOR 值是有效的。
 - CFM_FACE——szFaceName 字段的值是有效的。
 - CFM_ITALIC——dwEffects 字段中的 CFE_ITALIC 值是有效的。
 - CFM_OFFSET——yOffset 字段是有效的。
 - CFM_PROTECTED——dwEffects 字段中的 CFE_PROTECTED 值是有效的。
 - CFM_SIZE——yHeight 字段是有效的。
 - CFM_STRIKEOUT——dwEffects 字段中的 CFE_STRIKEOUT 值是有效的。
 - CFM_UNDERLINE——dwEffects 字段中的 CFE_UNDERLINE 值是有效的。

- dwEffects——字符效果，可以是以下值的组合：
 - CFE_AUTOCOLOR——使用系统正文颜色。
 - CFE_BOLD, CFE_ITALIC, CFE_STRIKEOUT 和 CFE_UNDERLINE——粗体字符、斜体字符、带删除线和带下划线。
 - CFE_PROTECTED——字符是受保护的，企图改变字符的话，控件会向父窗口发送一个 EN_PROTECTED 通知消息。
- yHeight——字符高度，单位是 1/1 440 英寸（或 1/20 磅），如果这里是 180，换算到“字体选择”通用对话框中的尺寸就是 9 磅（ $180 \times 1/20 = 9$ ）。
- yOffset——从基线算起的字符偏移，单位同上，如果该成员是正值，字符显示为上标；如果是负值，字符显示为下标。
- crTextColor——正文颜色，如果在 dwEffects 字段中指定了 CFE_AUTOCOLOR 标志，那么这个值会被忽略。
- bCharSet 和 bPitchAndFamily——字符集。
- szFaceName——用字符串表示的字体名字。

通过填充这个结构并将它通过 EM_SETCHARFORMAT 消息传送给控件，可以改变文字的效果（粗体、斜体、带删除线、带下划线等），如正文颜色（crTextColor）、字体外观（szFaceName）、字体大小（yHeight），以及使用的字符集等。使用 CHARFORMAT2 结构可以设置更多的文本风格，如字间距与正文背景色等。如果不需要这些额外的功能，那么只要使用 CHARFORMAT 结构就可以了。

例子程序 Richedit.asm 中只演示了使用 Plain Text 模式为所有文本设置字体的方法：

```

szFont    .const
          db      '宋体',0
          .code
          invoke   SendMessage,hWinEdit,EM_SETTEXTMODE,TM_PLAINTEXT,0
          invoke   RtlZeroMemory,addr @stCf,sizeof @stCf
          mov      @stCf.cbSize,sizeof @stCf
          mov      @stCf.yHeight,9 * 20
          mov      @stCf.dwMask,CFM_FACE or CFM_SIZE or CFM_BOLD
          invoke   lstrcpy,addr @stCf.szFaceName,addr szFont
          invoke   SendMessage,hWinEdit,EM_SETCHARFORMAT,0,addr @stCf

```

程序首先将控件的模式设置为 Plain Text 模式，然后定义了一个名为@stCf 的 CHARFORMAT 结构，将结构长度设置为 CHARFORMAT 结构的长度，然后在 dwMask 字段中使用 CFM_FACE，CFM_SIZE 和 CFM_BOLD 标志，表示只使用字体、字体大小和字体粗细参数，最后发送 EM_SETCHARFORMAT 消息将控件中的全部文字设置为大小为 9 磅（小五号）的“宋体”。

4. 装入和保存文本

显然，使用 GetWindowText 和 SetWindowText 函数来保存和装入文本是可行的，但是 RichEdit 控件支持很大的文件，当文件足够大的时候，使用这种方法就很麻烦，因为必须首先

要分配一块足够大的内存用做缓冲区，为了解决这个问题，Richedit 控件提供了一种新方法，那就是文本流（Text Streaming）。

考虑这样一种情况：为了装入文本，可以申请一块大小合适的缓冲区（当然不会大到能容纳全部文本），每次从文件中读入缓冲区大小的文本并添加到控件中，如此循环直到读入全部文本；保存文本的时候，同样可以每次从控件中取出缓冲区大小的文本，然后写入文件，如此循环直到处理完控件中的全部文本。

文本流的操作方法与此类似，只不过缓冲区和循环都封装在控件的内部，而我们通过提供回调函数的办法提供读写文件的功能模块，Richedit 控件循环调用这个模块直到处理完全部的内容。每次调用的时候，控件通过参数告诉回调函数要读写的字节数和缓冲区的地址。文本的流入和流出使用 EM_STREAMIN 和 EM_STREAMOUT 消息，这两个消息的使用格式是一样的：

invoke	SendMessage, hWinEdit, EM_STREAMIN, uFormat, lpStream
invoke	SendMessage, hWinEdit, EM_STREAMOUT, uFormat, lpStream

wParam 参数中的 uFormat 指定需要流入/流出的内容，它可以是以下的取值：

- SF_RTF——文本格式是 RTF 格式。
- SF_TEXT——文本是 Plain Text 格式，也就是简单的文本格式。
- SFF_SELECTION——流操作的范围是当前选择区域。如果将文本流入，当前选择区域就会被替换；如果是流出，则只有那些当前选定的文本才流出。如果没有指定这个标志，操作范围是控件中的所有文本。
- SF_UNICODE——指定的是 Unicode 文本（2.0 及以上版本提供）。

lParam 参数中的 lpStream 指向一个 EDITSTREAM 结构，该结构定义如下：

EDITSTREAM	STRUCT
dwCookie	DWORD ? ;用户自定义值
dwError	DWORD ? ;用来返回流操作过程中的错误信息
pfnCallback	DWORD ? ;回调函数地址
EDITSTREAM	ENDS

结构中各字段的说明如下：

- dwCookie——应用程序自定义的数值，这个数值将会传递给回调函数。
- dwError——指示流操作的结果，0 说明没有错误。
- pfnCallback——指向回调函数，该函数由用户定义，并由 RichEdit 控件调用来传输文本。RichEdit 控件将文本分成多个部分，每次调用该函数处理一个部分，直到全部文本被处理为止。

回调函数的定义如下：

EditStreamCallback	proc dwCookie, lpBuffer, NumBytes, pBytesTransferred
--------------------	--

控件传递给回调函数的参数说明如下：

- dwCookie——就是消息中指定的 EDITSTREAM 结构中定义的 dwCookie 值。

TRUE 还是 FALSE 来决定进行读文件还是写文件的操作。

存盘时，程序使用流出操作，这时控件会将所有的文本输出一遍，如果不将原文件中的内容清除，那么输出的文本就会叠加在原始文件后面，这显然不是我们需要的结果，所以程序使用 SetFilePointer 和 SetEndOfFile 首先将原文件清空（这两个函数以及读写文件的函数 ReadFile 和 WriteFile 的用法请参考第 10 章的 10.2.2 小节）。

5. 查找和替换

在 Richedit 控件中可以通过发送 EM_FINDTEXT 或者 EM_FINDTEXTX 消息来完成查找字符串的功能，EM_FINDTEXTX 消息是 EM_FINDTEXT 消息的扩展，它们的用法是：

invoke	SendMessage, hWinEdit, EM_FINDTEXT, uFlags, lpFindText
invoke	SendMessage, hWinEdit, EM_FINDTEXTX, uFlags, lpFindTextEx

消息的 wParam 参数中的 uFlags 指定查找的选项，它可以是以下取值的组合：

- FR_DOWN（2.0 版本及以上使用）——向后查找，不设置的话表示向前查找。
- FR_MATCHCASE——查找字符串区分大小写。
- FR_WHOLEWORD——匹配整个单词。

EM_FINDTEXT 的 lParam 参数指向一个 FINDTEXT 结构，而 EM_FINDTEXTX 消息的 lParam 参数指向一个 FINDTEXTX 结构，这两个结构的定义如下：

```

FINDTEXT STRUCT
    chrg          CHARRANGE <> ;查找区域
    lpstrText     DWORD      ?;查找字符串地址
FINDTEXT ENDS
FINDTEXTX STRUCT
    chrg          CHARRANGE <>;查找区域
    lpstrText     DWORD      ?;查找字符串地址
    chrgText      CHARRANGE <>;如果找到则在这里返回找到文字的起始/结束位置
FINDTEXTX ENDS

```

可以看出，EM_FINDTEXTX 消息的扩展之处在于直接在 chrgText 字段中返回找到文字的区域，程序可以马上使用这个区域数据进行其他操作，而 EM_FINDTEXT 消息必须根据找到的位置和查找字符串的长度自己计算这个区域。

两个消息的返回值是一样的，如果没有找到指定文字则返回-1，否则返回找到文字的起始位置。例子程序在 FindText 子程序中完成查找功能，这个子程序分别在“查找下一个”、“查找上一个”和“查找文字”通用对话框的自定义消息中被调用，子程序中通过下面的代码来设置查找区域：

```

invoke    SendMessage, hWinEdit, EM_EXGETSEL, 0, addr @stFindText.chrg
.if      stFind.Flags & FR_DOWN
    push    @stFindText.chrg.cpMax
    pop     @stFindText.chrg.cpMin
.endif
mov      @stFindText.chrg.cpMax, -1

```

当找到一个匹配字符串后，字符串被设置为选择区域，如果向下继续查找下一个的话，必须将这个选择区域的结束位置用做下一次查找的起始点，所以程序发送 EM_EXGETSEL 消息获取选择区域并将 cpMax 字段放到 cpMin 字段中，并将 cpMax 字段设置为 -1，表示一直查找到全部文本的最后。

在第 8 章中已经有所介绍：“查找文字”通用对话框使用 FINDREPLACE 结构，不知道是巧合还是 Microsoft 的故意安排，FINDREPLACE 结构中的 Flags 字段和 EM_FINDTEXT 消息中的 wParam 参数的标志定义是一样的，所以程序直接取出 Flags 字段，然后屏蔽掉 FR_MATCHCASE, FR_DOWN 和 FR_WHOLEWORD 等不需要使用的标志以后，就可以直接在消息的 wParam 参数中使用了：

```

mov     ecx, stFind.Flags
and     ecx, FR_MATCHCASE or FR_DOWN or FR_WHOLEWORD
invoke  SendMessage, hWinEdit, EM_FINDTEXT, ecx, addr @stFindText

```

找到字符串以后，进行替换操作就不是一件复杂的事情了，因为 FINDTEXT 结构的 chrgText 字段中已经返回了找到文本的起始和结束位置，将它设置为选择区域以后就可以通过发送 EM_REPLACESEL 消息进行替换操作了。

9.4.3 Richedit 控件的通知消息

Richedit 控件也可以向父窗口发送多种通知消息，使用“可以”一词的意思是，控件在默认状态下并不发送通知消息，如果需要控件发送某个消息，必须首先对控件进行设置。通过向 Richedit 控件发送 EM_SETEVENTMASK 消息可以激活需要的通知消息：

```

invoke  SendMessage, hWinEdit, EM_SETEVENTMASK, 0, dwMask

```

dwMask 是事件掩码，可以是下面标志的组合，分别代表激活不同的通知消息：

- ENM_CHANGE——允许 EN_CHANGE 通知码，本消息在用户的操作可能改变控件中的文本的时候发送。
- ENM_CORRECTTEXT——允许 EN_CORRECTTEXT 通知码。
- ENM_DRAGDROPDONE——允许 EN_DRAGDROPDONE 通知码，本消息在用户完成了一个拖放操作后发送。
- ENM_DROPFILES——允许 EN_DROPFILES 通知码，本消息在用户将一个文件拖放进控件后发送。
- ENM_KEYEVENTS——允许为键盘消息发送 EN_MSGFILTER 通知码。
- ENM_MOUSEEVENTS——允许为鼠标消息发送 EN_MSGFILTER 通知码。
- ENM_PROTECTED——允许发送 EN_PROTECTED 通知码。
- ENM_SCROLL——允许发送 EN_HSCROLL 和 EN_VSCROLL 通知码。
- ENM_SCROLLLEVENTS——允许为鼠标滑轮发送 EN_MSGFILTER 通知码。

- ENM_SELCHANGE——允许发送 EN_SELCHANGE 通知码，本消息在选择区域改变以后发送（包括光标位置改变）。
- ENM_UPDATE——允许发送 EN_UPDATE 通知码，本消息在控件将要显示被改变的文本之前发送。

当这些通知消息被激活的时候，父窗口就可以收到包含相应通知码的 WM_NOTIFY 消息。一般来说，将控件设置为仅发送程序感兴趣的通知消息，如当在 Richedit 控件中按下右键需要弹出一个菜单的时候，需要检测鼠标消息，那么就需要指定 ENM_MOUSEEVENTS 标志；另外，如果需要随时检测选择区域的状态，以便随时设置工具栏中“拷贝”与“剪切”等按钮的状态，那么就要使用 ENM_SELCHANGE 标志，这样光标一移动或者选择区域一改变，父窗口就可以收到 EN_SELCHANGE 通知码，于是程序就可以在 WM_NOTIFY 消息中随时改变工具栏上各按钮的状态。

由于 Richedit.asm 例子中没有使用工具栏，所以没有对通知消息进行示例，有兴趣的读者可以查看所附光盘的 Chapter09\Wordpad 目录中的例子文件，这个例子中使用工具栏和状态栏等控件，需要随时显示光标位置等信息，程序中就包含了处理通知消息的代码。

9.5 窗口的子类化

9.5.1 什么是窗口的子类化

在使用控件的过程中，常常会遇到这样一种情况：我们需要一种窗口，它与某种现成的控件提供的功能很相似，如果使用现成控件的话，那么控件几乎能提供所有需要的功能，仅我们要求的某个细节无法实现。举例来说，要编写一个十六进制与十进制的转换程序，程序中需要两个编辑控件来输入数值，输入十进制数值可以使用现成的 Edit 控件，只要指定 ES_NUMBER 风格就能让编辑框只能输入数字 0~9，但输入十六进制数值的时候就不行了，因为指定 ES_NUMBER 风格的话就无法输入 A~F，不指定的话用户就可能输入 0~9 和 A~F 之外的字符，那么该如何处理呢？

解决的办法有两种，第一种当然是自己创建一个窗口类，然后在自己的窗口过程中完成所有的功能，这显然是一项费时又费力的工作，因为我们几乎要自己重新写一遍 Edit 控件的全部功能；第二种方法就是使用本节要介绍的窗口子类化，窗口子类化最适合做的就是这一类工作。

如图 9.8 所示，窗口子类化的含义是接管被子类化的控件窗口，以达到对它进行控制的目的。虽然控件的窗口过程被封装在 Windows 内部，无法对它进行直接修改，但只要能截获 Windows 给控件的窗口过程发送的消息，就能够控制控件窗口。以上面的要求为例，只要截获 Windows 向编辑控件发送的 WM_CHAR 消息，就能够根据需要丢弃包含非十六进制字符的 WM_CHAR 消息，只把包含十六进制字符的 WM_CHAR 转发给控件的窗口过程，这样编辑控件将根本收不到十六进制字符之外的字符，我们的要求也就达到了。

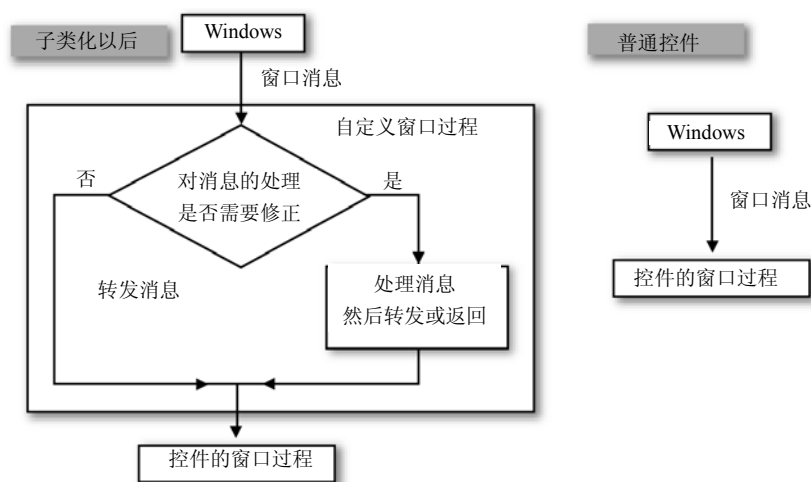


图 9.8 窗口子类化的工作原理

子类化的操作并不局限于控件窗口，实际上任何窗口都可以子类化。但是对于应用程序自身使用的窗口类来说，它的控制权本来就是 100% 属于应用程序自身的，要实现某种功能就直接修改源代码好了，没有必要再进行一个子类化的过程，所以子类化的操作往往是对“黑匣子”类型的控件窗口进行的。

9.5.2 窗口子类化的实现


窗口子类化的要点是截获窗口的窗口过程，如何实现这一点呢？每个窗口的内部都保存有它所属的窗口类的 `WNDCLASSEX` 结构，结构中的 `lpfnWndProc` 字段指出了窗口过程的地址，如果能用自己的窗口过程地址来替换这个地址，那么 Windows 就会把消息发送到自定义的窗口过程中来了。通过调用函数 `SetWindowLong` 可以实现这个功能，`SetWindowLong` 函数的用法是这样的：

invoke	SetWindowLong, hWnd, nIndex, dwNewLong
mov	dwOldLong, eax

`hWnd` 参数指定要子类化窗口的窗口句柄，`nIndex` 参数指定需要修改窗口的哪个属性，它可以是以下的取值：

- `GWL_EXSTYLE`——窗口的扩展风格。
- `GWL_STYLE`——窗口风格。
- `GWL_WNDPROC`——窗口过程地址（这就是我们需要的）。
- `GWL_HINSTANCE`——窗口所属的模块实例句柄。
- `GWL_ID`——窗口 ID。
- `GWL_USERDATA`——窗口附带的 32 位自定义数值。

`dwNewLong` 参数指定新的属性值。如果 `nIndex` 为 `GWL_WNDPROC`，`dwNewLong` 表示新的窗口



```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#include                <resource.h>
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#define    ICO_MAIN            1000
#define    DLG_MAIN            1000
#define    IDC_HEX              1001
#define    IDC_DEC              1002
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN                ICON "Main.ico"
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
DLG_MAIN DIALOG 107, 102, 129, 42
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Hex <> Dec"
FONT 9, "宋体"
{
    LTEXT "Hex", -1, 7, 9, 15, 8
    EDITTEXT IDC_HEX, 27, 7, 94, 12
    LTEXT "Dec", -1, 7, 26, 15, 8
    EDITTEXT IDC_DEC, 27, 24, 94, 12, ES_NUMBER
}
```

汇编源程序 SubClass.asm 的内容如下:

[illegible]

318

[illegible]

```

;*****
    .elseif    eax ==    WM_COMMAND
        mov     eax, wParam
        .if     ! dwOption
            mov     dwOption, TRUE
            .if     ax ==    IDC_HEX
                invoke    _HexToDec
            .elseif    ax ==    IDC_DEC
                invoke    _DecToHex
            .endif
            mov     dwOption, FALSE
        .endif
    .else
        mov     eax, FALSE
        ret
    .endif
    mov     eax, TRUE
    ret

_ProcDlgMain    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
start:
        invoke    GetModuleHandle, NULL
        mov     hInstance, eax
        invoke    DialogBoxParam, hInstance, DLG_MAIN, \
            NULL, offset _ProcDlgMain, NULL
        invoke    ExitProcess, NULL
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    end        start

```

程序运行后将显示如图 9.9 所示的对话框，上面的编辑控件是经过子类化的，只能输入十六进制字符，当输入字母 a~f 的时候，不管输入的是大写还是小写字母，都会被控件转换成大写字母。下面的编辑框是 ES_NUMBER 风格的，可以输入数字。不管在哪个编辑框中输入数值，

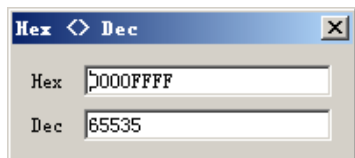


图 9.9 窗口子类化例子的运行界面

程序会马上进行转换并将结果在另一个编辑框中显示出来。

程序在对话框的初始化消息 WM_INITDIALOG 中对这两个编辑控件发送 EM_LIMITTEXT 消息，以此限制能够输入的最大字符长度，接下来程序通过 GetDlgItem 获取 IDC_HEX 编辑框的窗口句柄，并使用 SetWindowLong 函数将编辑框的新窗口过程设置到 _ProcEdit 子程序中，返回

的原窗口过程地址被保存到 lpOldProcEdit 变量中：

```

    invoke    GetDlgItem, hWnd, IDC_HEX
    invoke    SetWindowLong, eax, GWL_WNDPROC, addr _ProcEdit
    mov     lpOldProcEdit, eax

```

这样，当 Windows 向 IDC_HEX 编辑框发送消息时，_ProcEdit 子程序就会收到消息，在 _ProcEdit 中我们仅处理感兴趣的 WM_CHAR 消息，程序在数据段中定义了一个允许输入的字符表，表中包括 0~9、大小写的 A~F，以及退格键（如果不允许输入退格键的话将无法修正输

入错误)。然后在处理 WM_CHAR 消息时使用 scasb 指令进行查表,如果字符在表中,则将 WM_CHAR 消息通过 CallWindowProc 函数转发给原来的窗口过程,如果不在表中则直接返回,相当于丢弃了这个按键动作。代码如下:

```

        .const
szAllowedChar    db      '0123456789ABCDEFabcdef', 08h
        .code

        ...

        mov     eax,wParam    ; WM_CHAR 消息中的一段代码
        mov     edi,offset szAllowedChar
        mov     ecx,sizeof szAllowedChar
        repnz   scasb
        .if     ZERO?
        .if     al > '9'
                and     al,not 20h
        .endif
        invoke  CallWindowProc, lpOldProcEdit, hWnd, uMsg, eax, lParam
        ret
    .endif

```

在转发之前,程序还对字符进行判断,如果字符是小写的 a~f 的话(表中的字符中大于“9”的肯定是字母),则通过 and al, not 20h 语句将字母转换成大写,因为大写字母的 ASCII 码从 41h 开始,小写字母从 61h 开始,这样的计算方法对大写字母没有影响,对小写的则刚好能够转换成大写的。程序将其他所有的消息原封不动地转发给原来的窗口过程,这样才能让编辑控件的窗口过程为我们完成控件的其他功能。

转发消息使用了 CallWindowProc 函数,这个函数仅起到将参数入栈和调用指定地址的作用,对于下面的语句我们完全可以用自己调用 lpOldProcEdit 的方法来代替它:

```
invoke    CallWindowProc, lpOldProcEdit, hWnd, uMsg, eax, lParam
```

下面的代码就可以完成同样的功能:

```

push     lParam
push     eax
push     uMsg
push     hWnd
call     lpOldProcEdit

```

程序中的其他代码应该算是相当简单的, _DecToHex 子程序是十进制到十六进制的转换子程序,子程序中用 GetDlgItemInt 函数读入编辑框中的十进制数值,并用 wsprintf 转换成十六进制数值的字符串并显示到 IDC_HEX 编辑框中; _HexToDec 是十六进制到十进制的转换子程序,由于并没有现成的转换函数,所以在子程序中顺序读入字符并每次通过乘以 16 来进行计算。



对控件窗口进行子类化,影响的只是被操作的窗口,并不会影响基于这种控件的其他窗口,因为 SetWindowLong 函数操作的对象只是单个窗口而不是窗口类,所以要对多个控件窗口进行同样的子类化就必须对每个窗口都进行子类化操作。

程序中还有一个技巧。由于使用 SetDlgItemText 设置编辑框文本的时候,编辑框会发送

WM_COMMAND 消息，又由于一收到某个 WM_COMMAND 消息就进行转换计算，并再次使用 SetDlgItemText 函数将计算结果显示在另一个编辑框中，这样就会进入发送 WM_COMMAND 消息的死循环中。为此程序中定义了一个 dwOption 变量，当正在处理某个 WM_COMMAND 消息的时候，将这个变量设置为 1 来防止重入，这样就能够防止死循环的发生，代码如下：

```

    .elseif    eax ==    WM_COMMAND
        mov     eax, wParam
        .if     ! dwOption
            mov     dwOption, TRUE
            .if     ax ==    IDC_HEX
                invoke    _HexToDec
            .elseif  ax ==    IDC_DEC
                invoke    _DecToHex
            .endif
            mov     dwOption, FALSE
        .endif
    .endif

```

9.6 控件的超类化

9.6.1 什么是控件的超类化

子类化是对窗口功能的调整和扩展，那么超类化是什么呢？超类化是对类的调整和扩展，在 C++ 中，可以通过继承和扩展某个基类来形成一个派生的类，超类化可以完成的功能与这相似。

超类化主要用在什么地方呢？举例来说，如果需要一个只能输入十六进制字符的编辑框，那么可以通过对编辑框窗口子类化来实现，9.5 节的例子就是如此，但是当应用程序需要大量使用这种十六进制编辑框时，该如何处理呢？方法有 3 种：

- 创建自己的类，自己书写所有的功能代码。
- 创建多个 Edit 控件，并把它们全部子类化。
- 超类化 Edit 控件，用 Edit 控件当做基类派生出一个新的类，并用这个类来建立多个“新 Edit”控件窗口。

第一种方法在 9.5 节中就被“枪毙”了，几乎没有人去干这种吃力不讨好的事情；第二种方法要好一点，但子类化一大堆的控件也是一件令人头痛的事情；这时就应该使用超类化 Edit 类方法，当从 Edit 类派生出一个新的“十六进制编辑类”后，接下来直接使用这个类就能够创建出一大堆的十六进制编辑框。

9.6.2 控件超类化的实现

各种自定义的窗口和不同的控件窗口之所以看上去千姿百态，功能也各不相同是因为两个原因：首先用来表示类属性的 WNDCLASSEX 结构定义不同，造成窗口的风格与形状等各不相同；其次，不同窗口类使用的窗口过程不同，这些不同的窗口过程对各种消息的处理方法各不相同，造成窗口的功能不同。

哭

invoke	GetClassInfoEx, hinst, lpszClass, lpwcx
--------	---

[illegible]

读者可以看到,对话框中定义了多个 HexEdit 类,但是系统中并没有预定义这种名称的类,这就是将要 Edit 类中派生的类。SuperClass.asm 文件的内容如下:

324

哭

由于在对话框初始化的时候，对话框管理器就要根据对话框资源的内容创建每个子窗口控件，所以在调用 `DialogBoxParam` 函数显示对话框之前，“HexEdit”类就必须存在，否则初始化工作会失败。因此，程序在 `DialogBoxParam` 函数之前调用 `SuperClass` 子程序进行超类化的工作。

```

        .const
szEditClass    db      'Edit',0
szClass        db      'HexEdit',0
               .code
               ...
mov            @stWC.cbSize, sizeof @stWC ; @stWC 是一个 WNDCLASSEX 结构
invoke        GetClassInfoEx, NULL, addr szEditClass, addr @stWC
push @stWC.lpfWndProc
pop           lpOldProcEdit
mov           @stWC.lpfWndProc, offset _ProcEdit
push hInstance
pop           @stWC.hInstance
mov           @stWC.lpszClassName, offset szClass
invoke        RegisterClassEx, addr @stWC

```

本程序演示的是派生类在对话框中的使用情况，在这里基于派生类创建的窗口是由对话框管理器自动调用 `CreateWindowEx` 函数创建的，如果将派生类使用在普通窗口中的话，可以通过指定派生类的名称，自己使用 `CreateWindowEx` 函数来创建。

第 10 章

内存管理和文件操作

本章涉及内存的分配、释放，以及文件 I/O 的有关操作，之所以把这两部分集中在一章中，是因为它们的关系比较密切——进行与文件有关的操作之前往往需要分配内存，以使用做文件读写的缓冲区，而要将内存中的数据保存起来往往需要写文件。

Windows 操作系统中有一个新的概念：内存映射文件。通过这个功能将文件的内容直接映射到内存中，并使用读写内存的方法来对文件进行读写，内存映射文件是内存管理函数的一种，但它必须与文件操作函数配合使用。

本章的 10.1 节介绍与内存管理相关的内容，接下来介绍与文件、目录、磁盘操作有关的内容。由于内存映射文件要涉及文件操作，所以放在最后一节讨论。

10.1 内存管理

10.1.1 内存管理基础

Win32 中的内存管理是分层次的，系统提供了几组层次不同的函数来管理内存，它们是标准内存管理函数、堆管理函数、虚拟内存管理函数和内存映射文件函数。所有的这些函数都是为了让用户能在比较高的层次上方便地管理内存，以便将程序和底层的内存分页机制隔离开来。如图 10.1 所示，这几组函数的层次是各不相同的。

Windows 使用一个以页为基础的虚拟内存系统，与分页有关的概念已经在第 1 章的 1.3.2 节中有所介绍。Windows 充分利用了 80x86 处理器保护模式下的线性寻址机制和分页机制，这些机制是 Win32 内存管理的基础。Win32 提供了一组虚拟内存管理函数来管理虚拟内存，主要用于保留/提交/释放虚拟内存，在虚拟内存页上改变保护方式、锁定虚拟内存页，以及查询一个进程的虚拟内存等操作，这是一组位于底层的函数。

堆管理函数相对比较高级一点，堆的主要功能就是有效地管理内存和进程的地址空间。DOS 操作系统下的 C 语言中就已经有了“堆”的概念，这时的“堆”是程序初始化时向操作系统申请并预留的大块内存，程序通过 C 函数在这块空间中申请和释放内存。

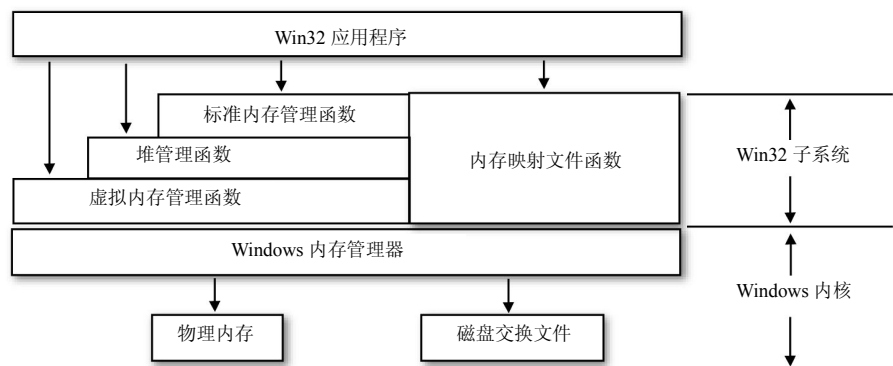


图 10.1 Windows 的内存分层管理

在 Win32 中，进程可以使用的整个地址空间就是一个堆。并且“堆”的概念又被引申了一步：Win32 中分两种堆，一种是进程的“默认堆”，默认堆只有一个，指的就是可以使用的整个地址空间；另一种是“动态堆”，也称为“私有堆”，私有堆类似于 DOS 下 C 语言中使用的那种堆，一个进程可以随意建立多个私有堆，也可以随意将它们释放，私有堆全部位于默认堆中，从概念上看，它和默认堆并没有什么不同，就像一个跨国公司和属下的子公司同样都是按照公司的规程操作一样。使用堆管理函数可以对所有的私有堆和默认堆进行操作。

标准内存管理函数总是在默认堆中分配和释放内存，这组函数就是常规意义上的内存管理函数。

内存映射文件函数相对比较独立，它是为了文件操作的方便性而设立的，当对文件进行操作的时候，一般总是先打开文件，然后申请一块内存用做缓冲区，再将文件数据循环读入并处理，当文件长度大于缓冲区长度的时候需要多次读入，每次读入后处理缓冲区边界位置的数据往往是个麻烦的问题。曾经介绍过 Windows 可以使用磁盘文件当做虚拟内存（参考图 1.5，虚拟内存的实现），内存映射文件函数使用同样的办法将一个文件直接映射到进程的地址空间中，这样可以通过内存指针用读写内存的办法直接存取文件内容。

对比这些函数，可以发现它们涉及的系统资源是各不相同的，如表 10.1 所示。

表 10.1 不同内存管理函数的操作对象

内存管理函数	涉及方面
标准内存管理函数	一个进程的默认堆
堆管理函数	一个进程的虚拟地址空间、系统内存、进程堆资源结构
虚拟内存管理函数	一个进程的虚拟地址空间、系统页文件、系统内存、硬盘空间
内存映射文件函数	一个进程的虚拟地址空间、系统页文件、系统内存、硬盘空间、标准文件 I/O

10.1.2 内存的当前状态

在第 1 章中已经介绍过，一个进程可以寻址的地址空间是 4 GB，但用户可以直接管理的地址空间是多大呢？实际上，高端的 2 GB 是供操作系统内核使用的，其中安排了操作系统的代码和数据（Windows 9x 中还包括共享内存映射的地址空间），可供应用程序使用的地址空间是

低端的 2 GB，这 2 GB 除去应用程序与用户 DLL 等的代码和静态数据段以后，余下来的才是内存管理函数可以使用的地址空间，应用程序和用户 DLL 的大小一般只有几兆字节到几十兆字节，所以可以认为能自由使用的地址空间基本上是 2 GB。

既然用户可以使用的地址空间大概为 2 GB，读者千万不要认为就可以申请 2 GB 的内存了，因为这 2 GB 只是可以使用的“地址”空间，而不是可以使用的“内存”空间，可分配内存的大小还受制于物理内存和磁盘交换文件的大小。因为物理内存和磁盘交换文件是供整个系统和所有用户程序使用的，所有系统内核、当前执行的所有用户程序的代码、数据，以及分配的内存总量并不能超过物理内存和磁盘交换文件的总和。

当设计一个可能需要申请大量内存的程序时，如何预先得知系统的配置情况呢？对此可以使用 GlobalMemoryStatus 函数：

invoke	GlobalMemoryStatus, lpBuffer
lpBuffer 指向一个 MEMORYSTATUS 结构，结构的定义如下：	
MEMORYSTATUS STRUCT	
dwLength	DWORD ? ;本结构的长度
dwMemoryLoad	DWORD ? ;已用内存的百分比
dwTotalPhys	DWORD ? ;物理内存总量
dwAvailPhys	DWORD ? ;可用物理内存
dwTotalPageFile	DWORD ? ;交换文件总的大小
dwAvailPageFile	DWORD ? ;交换文件中空闲部分大小
dwTotalVirtual	DWORD ? ;用户可用的地址空间
dwAvailVirtual	DWORD ? ;当前空闲的地址空间
MEMORYSTATUS ENDS	

在调用之前需要首先将 dwLength 字段设置为 MEMORYSTATUS 结构的长度，当调用 GlobalMemoryStatus 函数后，函数会在结构中返回对应的数值。注意：dwTotalPageFile 字段返回的是交换文件的最大值，并不是当前实际建立的交换文件的大小，一般当前的交换文件大小会小于这个数值，但这个数值的大小也不是确定的，如果需要的话，系统会增加它的大小直到不再有空余的磁盘空间放置交换文件为止。

在所附光盘的 Chapter10\MemInfo 目录中的 MemInfo.asm 文件利用这个功能定时获取并显示当前内存的使用信息，部分源代码如下：

	.const		
szInfo	db	'物理内存总数	%lu 字节', 0dh, 0ah
	db	'空闲物理内存	%lu 字节', 0dh, 0ah
	db	'虚拟内存总数	%lu 字节', 0dh, 0ah
	db	'空闲虚拟内存	%lu 字节', 0dh, 0ah
	db	'已用内存比例	%d%%', 0dh, 0ah
	db	', 0dh, 0ah	
	db	'用户地址空间总数	%lu 字节', 0dh, 0ah
	db	'用户可用地址空间	%lu 字节', 0dh, 0ah, 0
	.code		
_GetMemInfo	proc		
	local	@stMemInfo:MEMORYSTATUS	
	local	@szBuffer[1024]:byte	

```
mov     @stMemInfo.dwLength, sizeof @stMemInfo
invoke  GlobalMemoryStatus, addr @stMemInfo
invoke  wsprintf, addr @szBuffer, addr szInfo, \
        @stMemInfo.dwTotalPhys, @stMemInfo.dwAvailPhys, \
        @stMemInfo.dwTotalPageFile, \
        @stMemInfo.dwAvailPageFile, \
        @stMemInfo.dwMemoryLoad, \
        @stMemInfo.dwTotalVirtual, @stMemInfo.dwAvailVirtual
invoke  SetDlgItemText, hWinMain, IDC_INFO, addr @szBuffer
ret

_GetMemInfo  endp
...
```

程序每隔 1 秒钟在 _GetMemInfo 子程序中用 GlobalMemoryStatus 获取内存状态并用 wsprintf 将数据转换成字符串，然后显示在对话框的 IDC_INFO 文本框中。



图 10.2 Meminfo 程序的运行结果

在笔者的计算机中，程序运行的结果如图 10.2 所示，计算机的物理内存配置为 128 MB，这个数值和物理内存总数（dwTotalPhys 字段）符合，dwMemoryLoad 字段为 72%，等于空闲物理内存（dwAvailPhys 字段）和物理内存总数的百分比与 100% 的差值，计算机上当前虚拟内存交换文件大小为 192 MB，小于最大限制 dwTotalPageFile 字段。

在与地址空间相关的数值上，dwTotalVirtual 字段的显示结果是 2 147 352 576，等于 2 GB 减去 128 KB，这是因为这 2 GB 的最低端的和最高端的两个 64 KB 是系统保留的（00000000h~0000ffffh，7fff0000h~7fffffffh）。

Windows 可以根据内存使用的需求自动调整交换文件的大小，比如，Meminfo 程序运行显示的当前磁盘交换文件可用空间（dwAvailPageFile 字段）为 168 MB，但是如果尝试申请一个高达 300 MB 的内存块，会发现仍然可以申请成功，这时 dwTotalPageFile 字段的大小会自动增长到 500 多兆字节，把内存块释放掉的话，dwTotalPageFile 字段会恢复到原来的数值。虚拟内存的使用给我们带来了很多的方便，我们可以使用超过物理内存好几倍的内存空间，但是如果所需的内存大大高于物理内存的大小，那么申请内存还是会失败，因为这会引起物理内存和交换文件之间的数据频繁交换，大量的磁盘请求将使系统性能降低到没有实际使用的意义，读者可以尝试在 128 MB 物理内存的计算机上申请一个 1 GB 的内存块，即使拥有远远大于 1 GB 的磁盘剩余空间供交换文件使用，也是不会成功的！如果读者需要使用大大高于物理内存的内存空间，可以尝试自己进行磁盘交换工作。

10.1.3 标准内存管理函数

标准内存管理函数的功能是在进程的默认堆中申请和释放内存块，它由下面一些函数组成：GlobalAlloc、GlobalFree 和 GlobalReAlloc 函数分别用来申请、释放和修改内存大小；GlobalLock 和 GlobalUnlock 函数用来进行锁定操作；而 GlobalDiscard，GlobalFlags，GlobalHandle 和 GlobalSize 等函数用来丢弃内存或获取已分配内存的一些信息。



这组函数是为了向后兼容而保留的，在 Win16 平台下，内存有“全局”和“本地”之分，全局堆（Global Heap）是系统中所有进程所共有的堆，包括系统进程及用户进程，它们使用上述 Global 开头的函数来使用全局堆中的内存。而每个进程又拥有一个私有的本地堆（Local Heap），进程使用一组 Local 开头的函数来使用本地堆中的内存，函数名称和上述 Global 开头的函数一一对应。

在 Win16 下，所有进程在全局堆中申请的内存会交错在一起，从而使得一个用户进程不小心的内存越界存取会导致整个操作系统的崩溃。到了 Win32 下，考虑到安全因素，可以被所有进程使用的全局堆被废弃，本地堆则改名为进程堆（Process Heap），也就是默认堆。

为了兼容性，GlobalAlloc/LocalAlloc 等函数被沿用了下来，但 Win32 下两组函数完全相同，它们都是在底层直接调用 HeapAlloc 从默认堆中分配内存。



由于 GlobalAlloc/LocalAlloc 等函数是为了兼容性目的而保留的，MSDN 中明确说明不建议继续使用，而是推荐使用下一节讲到的 HeapAlloc 等堆管理函数。

用标准内存管理函数可以分配的内存有两种：固定地址的内存块和可移动的内存块，而可移动的内存块又可以进一步定义为可丢弃的，让我们逐步来讨论它们的不同。

1. 固定的内存块

常规意义上的内存就是固定的内存块，因为申请到内存后，这块内存的线性地址是固定不变的。要申请一块固定的内存，可以使用函数：

```
invoke    GlobalAlloc,GMEM_FIXED or GMEM_ZEROINIT,dwBytes
.if      eax
mov      lpMemory,eax
.endif
```

第一个参数是标志，GMEM_FIXED 表示申请的是固定的内存块，GMEM_ZEROINIT 表示需要将内存块中的所有字节预先初始化为 0，也可以简单地使用 GPTR 标志，它就相当于是 GMEM_FIXED or GMEM_ZEROINIT；第 2 个参数 dwBytes 指出了需要申请的是以字节为单位的内存大小。如果内存申请失败，eax 中返回 NULL，否则返回值是一个指向内存块起始地址的指针，用户需要保存这个指针，在使用内存或者释放内存的时候还要用到它。

如果要释放一个先前申请的固定内存块，可以使用 GlobalFree 函数：

```
invoke    GlobalFree,lpMemory
```

如果释放成功，函数返回 NULL，否则函数返回的值就是输入的 lpMemory。程序在不再使用内存块的时候应该使用这个函数将内存释放，即使程序在退出的时候忘记了释放内存，Windows 也会自动将它们释放。

在实际使用中往往需要改变一个内存块的大小，这时候就要用到 GlobalReAlloc 函数，这个函数可以缩小或者扩大一块已经申请到的内存：

```
invoke    GlobalReAlloc,lpMemory,dwBytes,uFlags
.if      eax
```

```
        mov     lpNewMemory, eax
    .endif
```

lpMemory 是先前申请的内存块指针，dwBytes 是新的尺寸，如果这个数值比原来申请的时候要小，也就是需要缩小内存块，那么 uFlags 标志参数可以是 NULL。操作不成功时函数的返回值为 0，否则返回新的缩小了的内存块指针，当然，这个指针和原来的指针肯定是一样的。

但是需要扩大一个内存块的时候，情况就稍微有些复杂了。让我们做一个实验来模拟这样一种情况：首先申请两个 1000h 大小的固定内存块，得到两个指针，读者可以发现第二块几乎紧接第一块内存，一般情况下如果第一块内存的地址是 X，那么第二块内存的地址几乎就是 X + 1000h，如果需要将第一个内存块扩大到 2000h 字节，那么只能在别的地方开辟一个 2000h 大小的内存块，因为原来位置后面的 1000h 已经被第二块内存占用了，这就意味着新的指针可能和原来的不一样。

可以在 GlobalReAlloc 函数中通过指定不同的 uFlags 来规定是否允许 Windows 在必要的时候移动内存块。当 uFlags 中有 GMEM_MOVEABLE 选项的时候，如果需要移动内存块，Windows 会在别的地方开辟一块新的内存，并把原来内存块中的内容自动复制到新的内存块中，这时函数的返回值是新的指针，原来的指针作废。

如果不指定 GMEM_MOVEABLE 选项，那么只有当内存块后面扩展所需的空闲空间没有被使用时，函数才会执行成功，否则，函数失败并返回 NULL，这时原来的指针继续有效。

为了保证内存块扩大成功，建议总是使用下面的语句来扩大和缩小内存：

```
invoke GlobalReAlloc, lpMemory, dwBytes, GMEM_ZEROINIT or GMEM_MOVEABLE
.if     eax
    mov     lpMemory, eax
.endif
```

指定 GMEM_ZEROINIT 选项可以使内存块扩大的部分自动被初始化为 0，然后程序判断返回值，如果改变大小成功的话，则用新的指针替换原来的指针，其他和原来指针有关的值也不要忘了同时更新。

2. 可移动的内存块

可移动的内存块在不使用的时候允许 Windows 改变它的线性地址，为什么要使用可移动的内存块呢？惟一的理由是防止内存的碎片化，当进程长时间频繁地申请和释放不同大小的内存块后，申请的大量小块内存可能零散地分布在整个地址空间中，虽然空闲的内存总数不小，但是却没有剩下连续的大块空闲地址，导致无法再申请大块的内存。

解决内存碎片化的办法很简单，因为碎片之间有大量的内存是空闲的，只要允许 Windows 移动小块的在用内存，就可以将碎片合并成大块的空闲内存，但是在内存被移动后，程序中对应的指针也要随着改变，不然就会访问到错误的地址，而且，在使用内存的过程中，内存需要有个锁定的过程，否则用到一半的时候被 Windows 移动了，结果依然是错误的，只有程序将内存解锁，Windows 才可以自由移动它们，这就引申出可移动内存块的概念和操作的基本方法。

要申请一个可移动的内存块，使用的函数还是 GlobalAlloc，但需要使用不同的参数：

```

invoke    GlobalAlloc,GMEM_MOVEABLE or GMEM_ZEROINIT,dwBytes
.if      eax
mov      hMemory,eax
.endif

```

GMEM_MOVEABLE 标志指定了分配的内存是可移动的, GMEM_ZEROINIT 同样表示将申请到的内存块的内容初始化为 0 也可以用 GHND 标志,它就相当于 GMEM_MOVEABLE or GMEM_ZEROINIT); 如果内存申请失败, eax 中返回 NULL, 成功的话, 返回值是一个句柄而不是内存指针, 用户需要保存这个句柄, 在锁定或释放内存的时候还要用到它。一个进程可以申请的可移动内存的块数最大不能超过 65 536 个, 申请固定内存块时则没有数量限制。

要使用可移动内存之前, 需要把它锁定, 这相当于告诉 Windows 现在程序要使用这块内存了, 不能将它移动, 锁定内存使用 GlobalLock 函数:

```

invoke    GlobalLock,hMemory
.if      eax
mov      lpMemory,eax
.endif

```

函数的入口参数是 GlobalAlloc 返回的内存句柄, 如果锁定成功, 函数返回一个指针, 程序可以用使用固定内存块同样的方法来使用它; 如果锁定失败, 则函数返回 NULL。每次锁定返回的指针位置可能是不同的, 但内存块中的数据不会变化。

当程序暂时不需要操作这块内存的时候, 应该将它解锁, 否则和使用固定的内存块就没有区别了, 解锁使用 GlobalUnlock 函数:

```

invoke    GlobalUnlock,hMemory

```

函数的参数同样是 GlobalAlloc 返回的句柄, 解锁成功的话函数返回非 0 值。读者可能有个问题: 在多线程的程序中, 两个地方同时锁定内存, 但当一个地方还在使用的情况下另一个地方却调用 GlobalUnlock 将内存解锁了怎么办? 其实不用担心这个问题, Windows 为每个可移动的内存句柄维护一个锁定计数, 每次锁定内存的时候计数加 1, 解锁的时候计数减 1, 只有当计数为 0 的时候内存才真正被解锁, 所以只要程序中的 GlobalLock 函数和 GlobalUnlock 函数是配对的, 就不用担心这个问题。

要释放一个可移动的内存块, 同样使用 GlobalFree 函数:

```

invoke    GlobalFree,hMemory

```

但使用的参数是 GlobalAlloc 返回的内存句柄, 如果释放成功, 函数返回 NULL。不管内存当前是否处在锁定状态, 都可以被成功释放。


调整可移动内存块的大小, 同样使用 GlobalReAlloc 函数:

```

invoke    GlobalReAlloc,hMemory,dwBytes,GMEM_ZEROINIT or GMEM_MOVEABLE

```

如果调整成功, 返回值就是输入的 hMemory, 失败的话, 返回值是 NULL。即使内存块在锁定状态, 函数仍然可以调用成功, 但由于这时候内存块可能已经被移动了位置, 原来用 GlobalLock 函数获取的指针可能已经失效了, 所以调整可移动内存块的大小最好还是先将内存解锁, 等调整完毕以后再锁定使用。

 使用可移动内存块来防止内存碎片化的方法听起来很不错，但不幸的是，这只在 Win16 下有效。由于在 Win32 下上述函数仅仅是为了兼容性目的而存在的，系统仅仅保证这些函数在使用上不会出错，而在底层已经不再去做碎片拼合的工作了。而 HeapAlloc 等函数又不支持可移动内存块机制，所以在 Win32 下要防止内存碎片化，最好的办法还是使用内存池，有兴趣的读者可以自行查阅相关资料。

3. 可丢弃的内存块


分配可移动内存块的时候需要使用 GMEM_MOVEABLE 标志，如果同时配合使用 GMEM_DISCARDABLE 标志的话，生成的内存块是可丢弃的内存块，表示当 Windows 急需内存使用的时候，可以将它从物理内存中丢弃，可丢弃的内存块首先必须是可移动的内存块。函数调用如下：

```
invoke GlobalAlloc, GHND or GMEM_DISCARDABLE, dwBytes
.if eax
mov     hMemory, eax
.endif
```

当用 GlobalLock 锁定内存的时候，如果返回 NULL 指针，表示内存已经被 Windows 丢弃了，当然其中的数据也丢失了，程序需要重新生成数据。当内存块被丢弃的时候，内存句柄还是有效的，如果程序还要使用这个句柄，那么可以对它使用 GlobalReAlloc 函数来重新分配内存。

当可丢弃内存块的锁定计数为 0 时，程序也可以使用 GlobalDiscard 函数主动将它丢弃，这与 Windows 将它丢弃的效果是一样的：

```
invoke GlobalDiscard, hMemory
```

 使用内存函数时有两个地方需要特别注意：

（1）NULL 指针的检测——很多函数都返回内存指针，在使用指针前一定要检测它的有效性，如果使用了函数执行失败而返回的 NULL 指针来访问数据，会导致程序越权访问不该访问的地方，从而被 Windows 毫不留情地终止掉，这就是例子代码中总是有一个 if 语句来判断 eax 是否为 NULL 的原因。

（2）注意访问越界问题——越界操作也会引起越权访问，千万不要到超出内存块长度的地方去访问，例如，使用 lstrcpy 之类的函数处理字符串之前，先用 strlen 检测字符串长度是一个好习惯。

10.1.4 堆管理函数

Windows 的“堆”分为默认堆和私有堆两种。默认堆是在程序初始化时由操作系统自动创建的，所有的标准内存管理函数都是在默认堆中申请内存的；而私有堆相当于在默认堆中保留了一大块内存，用堆管理函数可以在这个保留的内存块中分配内存。一个进程的默认堆只有一个，而私有堆可以被创建多个。

默认堆可以直接被使用，而私有堆在使用前需要先创建，看上去要麻烦一点，但实际上，有些时候使用私有堆可能更有好处。

首先，可以使用默认堆的函数有多种，而它们可能不同的线程中同时对默认堆进行操作，为了保持同步，对默认堆的访问是顺序进行的，也就是说，在同一时间内每次只有一个线程能够分配和释放默认堆中的内存块。如果两个线程试图同时分配默认堆中的内存块，那么只有一个线程能够进行，另一个线程必须等待第一个线程的内存块分配结束之后才能继续执行。而私有堆的空间是预留的，不同线程在不同的私有堆中同时分配内存并不会引起冲突，所以整体的运行速度更快。

其次，当系统必须在物理内存和页文件之间进行页面交换的时候，系统的性能会受到很大的影响，在某些情况下，使用私有堆可以防止系统频繁地在物理内存和交换文件之间进行数据交换，因为将经常访问的内存局限于一个小范围地址的话，页面交换就不太可能发生，把频繁访问的大量小块内存放在同一个私有堆中就可以保证它们在内存中的位置接近。

再则，使用私有堆也有利于封装和保护模块化的程序。当程序包含多个模块的时候，如果使用标准内存管理函数在默认堆中分配内存，那么所有模块分配的内存块是交叉排列在一起的，如果模块 A 中的一个错误导致内存操作越界，可能会覆盖掉模块 B 使用的内存块，到模块 B 执行的时候出错了，我们却很难发现错误的源头来自于模块 A。如果让不同的模块使用自己的私有堆，那么它们使用的内存就会完全隔离开来，虽然越界错误仍然可能发生，但很容易跟踪和定位。

最后，使用私有堆也使大量内存的清理变得方便，在默认堆中分配的内存需要一块块单独释放，但将一个私有堆释放后，在这个堆里的内存就全部被释放掉了，并不需要预先释放堆中的每个内存块，这样非常便于模块的扫尾工作。

1. 私有堆的创建和释放

创建私有堆的函数是 HeapCreate:

```
invoke    HeapCreate, flOptions, dwInitialSize, dwMaximumSize
.if      eax && (eax < 0c0000000h)
mov       hHeap, eax
.endif
```

flOptions 参数是标志，用来指定堆的属性，可以指定的属性有 HEAP_NO_SERIALIZE 和 HEAP_GENERATE_EXCEPTIONS 两种。

HEAP_GENERATE_EXCEPTIONS 标志用来指定函数失败时的返回值，不指定这个标志的话，函

数失败时返回 NULL，否则返回一个具体的出错代码，以便于程序详细了解出错原因。出错代码的定义值都大于 0c0000000h，因为 0c0000000h 开始的地址空间为系统使用，分配的内存地址不可能高于这个地址，所以检测函数执行是否成功的时候可以使用上面的测试语句来比较返回值是否在 0~0c0000000h 之间。

HEAP_NO_SERIALIZE 标志用来控制对私有堆的访问是否要进行独占性的检测，前面曾经提到在默认堆中申请内存块的操作是顺序进行的，多个线程同时申请内存的请求只有一个能马上执行，其他将处于等待状态，对于一个私有堆来说，这个限制仍然存在，当从堆中分配内存时，系统有下面的操作步骤：

- (1) 遍历已分配的和空闲的内存块的链接表。
- (2) 寻找一个空闲内存块的地址。
- (3) 通过将空闲内存块标记为“已分配”来分配新内存块。
- (4) 将新内存块添加给内存块链接表。

当两个线程几乎同时在同一个堆中申请内存时，如果第一个线程执行了 (1)、(2) 两步的时候被系统切换到第二个线程，线程 2 同样又执行 (1)、(2) 两步，那么它们找到的空闲内存块就会是同一块内存，结果可想而知。解决问题的办法就是让单个线程独占对堆和它的链接表的访问权，当一个线程全部执行了这 4 个步骤后才允许第二个线程开始第一个步骤。

在用默认参数建立的堆中申请内存，系统会进行独占的检测工作，当然这要花费一定的系统开销。但是当以下情况存在时，可以保证不会同时有多个线程在同一个堆中申请内存：

- 进程只使用一个线程。
- 进程使用多个线程，但是每个线程只访问属于自己的私有堆。
- 进程使用多个线程，但程序中已经有其他措施来保证它们不会同时去访问同一个私有堆。

在这些情况下，可以指定 HEAP_NO_SERIALIZE 标志来建立私有堆，这样建立的堆不会进行独占性的检测，访问速度可以更快。

参数 dwInitialSize 指定创建堆时分配给堆的物理内存，随着堆中内存的分配，当这些内存被使用完时，堆的长度可以自动扩展。dwMaximumSize 参数指定了能够扩展到的最大值，当扩展到最大值时再尝试在堆中分配内存的话就会失败，这个值决定了系统给堆保留的连续地址空间的大小，函数会自动将这两个参数的数值调整为页面大小的整数倍。如果 dwMaximumSize 参数的值指定为 0，那么堆没有最大值限制，扩展范围只受限于空闲的内存总量。如果 dwMaximumSize 指定为非 0 值，在堆中申请的最大单个内存块不能大于 7FFF8h (相当于 524 KB)，dwMaximumSize 指定 0 的话就没有这个限制。

如果一个私有堆不再需要了，可以通过调用 HeapDestroy 函数将它释放：

invoke HeapDestroy, hHeap

释放私有堆可以释放堆中包含的所有内存块，也可以将堆占用的物理内存和保留的地址空间全部返还给系统。如果函数运行成功，返回值是 TRUE。当在进程终止的时候没有调用 HeapDestroy 函数将私有堆释放时，系统会自动释放。

虽然在默认堆中的内存申请主要使用标准内存管理函数，而堆管理函数的主要管理对象是私有堆，但是如果编程者愿意的话，也可以用堆管理函数在默认堆中分配内存，毕竟默认堆也是一个堆，但这样的话首先需要有一个句柄来代表默认堆，默认堆的句柄不能用 HeapCreate 来创建，但可以用 GetProcessHeap 函数来获取，这个函数没有输入参数，如果执行成功则返回默认堆的句柄。注意：因为这个句柄是“获取”的而不是“创建”的，所以不能调用 HeapDestroy 来释放它，如果对它调用 HeapDestroy 函数，系统会将它忽略。

2. 在堆中分配和释放内存块

如果要在堆中分配内存块，可以使用 HeapAlloc 函数：

```
invoke    HeapAlloc, hHeap, dwFlags, dwBytes
.if      eax && (eax < 0c0000000h)
    mov     lpMemory, eax
.endif
```

hHeap 参数就是前面创建堆时返回的堆句柄（或者使用 GetProcessHeap 函数得到的默认堆句柄），用来表示在哪个堆中分配内存，dwBytes 是需要分配的内存块的字节数，dwFlags 是标志，它可以是下面值的组合：

- HEAP_NO_SERIALIZE——当使用 HeapCreate 时指定了 HEAP_NO_SERIALIZE 标志，以后这个堆中使用的所有 HeapAlloc 函数都不进行独占检测。如果使用 HeapCreate 时没有指定 HEAP_NO_SERIALIZE 标志，可以在这里使用 HEAP_NO_SERIALIZE 标志单独指定对本次分配操作不进行独占检测。
- HEAP_GENERATE_EXCEPTIONS——如果申请内存失败函数返回具体的出错原因，而不仅返回一个 NULL。同样，当使用 HeapCreate 时指定了此标志的情况下，在这里就不必再一次指定。
- HEAP_ZERO_MEMORY——将分配的内存用 0 初始化。

当函数分配内存成功的时候，返回值是指向内存块第一个字节的指针，如果分配内存失败，返回值要视 dwFlags 的设置，如果没有指定 HEAP_GENERATE_EXCEPTIONS 标志，那么返回值为 NULL，否则，返回值可能是下面的数值：

- STATUS_NO_MEMORY——取值为 0C0000017h，表示内存不够。
- STATUS_ACCESS_VIOLATION——取值为 0C0000005h，表示参数不正确或者堆被破坏。

在堆中分配的内存块只能是固定地址的内存块，不像 GlobalAlloc 函数一样可以分配可移动的内存块。如果要释放分配到的内存块，可以使用 HeapFree 函数：

```
invoke    HeapFree, hHeap, dwFlags, lpMemory
```

hHeap 参数是堆句柄，lpMemory 是 HeapAlloc 函数返回的内存块指针，dwFlags 参数中也可以使用 HEAP_NO_SERIALIZE 标志，含义与使用 HeapAlloc 时相同。当函数执行成功的时候，

返回值为非 0 值，执行失败则函数返回 0。

对于用 HeapAlloc 分配的内存块，也可以使用 HeapReAlloc 重新调整大小：

```

invoke    HeapReAlloc, hHeap, dwFlags, lpMemory, dwBytes
.if      eax && (eax < 0c0000000h)
    mov     lpMemory, eax
.endif

```

其中 dwBytes 指定了新的大小，dwFlags 为标志，可以组合指定的标志有：

- HEAP_GENERATE_EXCEPTIONS——参见 HeapAlloc 函数的说明。
- HEAP_NO_SERIALIZE——参见 HeapAlloc 函数的说明。
- HEAP_ZERO_MEMORY——当扩大内存块的时候，将新增的部分初始化为 0，当缩小内存的时候，本参数无效。
- HEAP_REALLOC_IN_PLACE_ONLY——与 GlobalReAlloc 函数类似，当内存块的高处已经被其他内存块占据的时候，要扩大内存块必须将它移动位置，当没有指定这个标志的时候，函数会在需要的时候自动移动内存块，如果指定了这个标志，则不允许内存块移动，这时，当内存块高处不是空闲的时候，函数的执行会失败。

如果函数执行成功，返回值是指向新内存块的指针，显而易见，当缩小或扩大内存块时指定了 HEAP_REALLOC_IN_PLACE_ONLY 标志，则这个指针必定和原来的相同，否则的话，它既有可能和原来的指针相同也有可能不同。

3. 其他堆管理函数

除了上面的一些函数，堆管理函数中还有 HeapLock，HeapUnlock，GetProcessHeaps，HeapCompact，HeapSize，HeapValidate 和 HeapWalk 等函数。

GetProcessHeaps 函数用来列出进程中所有的堆（注意：不要和用来获取默认堆句柄的 GetProcessHeap 函数搞混），HeapWalk 用来列出一个堆中所有的内存块，HeapValidate 函数用来检验一个堆中所有内存块的有效性。这 3 个函数平时很少使用，一般在调试的时候使用。

GetProcessHeaps 函数的用法是：

```

invoke    GetProcessHeaps, NumberOfHeaps, lpHeaps

```

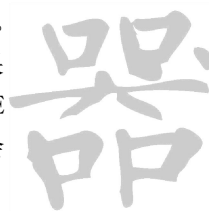
其中 lpHeaps 是一个指针，指向用来接收堆句柄的缓冲区，NumberOfHeaps 参数指定了这个缓冲区中可以存放句柄的数量，显然，缓冲区的长度应该等于 NumberOfHeaps 乘以 4 字节。函数执行后，进程中所有堆的句柄全部返回到缓冲区中，其中也包括默认堆的句柄。

HeapWalk 函数的用法是：

```

.repeat
    invoke    HeapWalk, hHeap, lpEntry
    push     eax
    ;检测缓冲区中的内存块信息
    pop      eax
.until      !eax

```



hHeap 是需要操作的堆句柄, lpEntry 指向一个包含有 PROCESS_HEAP_ENTRY 结构的缓冲区。调用 HeapWalk 函数时, 函数每次在 PROCESS_HEAP_ENTRY 结构中返回一个内存块的信息, 如果还有其他内存块, 函数返回 TRUE, 程序可以一直循环调用 HeapWalk 函数直到函数返回 FALSE 为止。在多线程的程序中使用 HeapWalk, 必须首先使用 HeapLock 函数将堆锁定, 否则调用会失败。

HeapValidate 用来验证堆的完整性或堆中某个内存块的完整性:

invoke	HeapValidate, hHeap, dwFlags, lpMemory
--------	--

其中 hHeap 指定要验证的堆。如果 lpMemory 为 NULL, 那么函数顺序验证堆中所有的内存块; 如果 lpMemory 指定了一个内存块, 则只验证这个内存块。dwFlags 是标志, 可以指定 HEAP_NO_SERIALIZE 标志。如果验证结果是所有的内存块都完好无损, 函数返回非 0 值, 否则函数返回 0。

HeapLock 函数和 HeapUnlock 函数用来锁定堆和解锁堆。这两个函数主要用于线程的同步, 当在一个线程中调用 HeapLock 函数时, 这个线程暂时成为这个堆的所有者, 也就是说只有这个线程能对堆进行操作 (包括分配内存、释放、调用 HeapWalk 等函数), 在别的线程中对这个堆的操作会等待在那里, 直到所有者线程调用 HeapUnlock 解锁为止。这两个函数的语法如下:

invoke	HeapLock, hHeap
invoke	HeapUnlock, hHeap

如果函数执行成功, 返回值为非 0 值, 否则函数返回 0。一般来说, 很少在程序中使用这两个函数, 而总是使用 HEAP_NO_SERIALIZE 标志来进行同步控制, 指定了这个标志的话, HeapAlloc, HeapReAlloc, HeapSize 和 HeapFree 等函数会在内部自己调用 HeapLock 和 HeapUnlock 函数。

HeapCompact 函数用于合并堆中的空闲内存块并释放不在使用中的内存页面:

invoke	HeapCompact, hHeap, dwFlags
--------	-----------------------------

HeapSize 函数返回堆中某个内存块的大小, 这个大小就是使用 HeapAlloc, 以及 HeapReAlloc 时指定的大小:

invoke	HeapSize, hHeap, dwFlags, lpMemory
--------	------------------------------------

lpMemory 指定了需要返回大小的内存块, 函数的返回值是内存块的大小, 如果执行失败, 函数返回-1。

10.1.5 虚拟内存管理函数

不管某个进程实际可用的物理内存是多少, 每个进程可以使用的地址空间总是 2 GB, 用户程序不必考虑一个线程地址对应的物理内存究竟安排在什么地方——是在真正的物理内存中? 在磁盘交换文件中? 还是根本没有物理内存与之对应。

一个进程的整个地址空间是客观存在的, 但是否有内存与该段地址空间中的地址相关联是

另外的问题，Windows 负责在适当的时间把线程地址映射到物理内存或磁盘上的交换文件上，这就是虚拟内存的基本概念。

在程序运行的时候，进程中每个地址都可以处于下列 3 种状态的 1 种中：

- 占用状态——线程地址已经映射到实际的物理内存中。也称为已提交状态。
- 自由状态——没有映射到物理内存中，线程地址当前也没有被程序使用。
- 保留状态——虽然线程地址没有映射到物理内存中，但它不会被使用，直到程序希望使用它为止。

进程开始的时候，所有地址都是处于自由状态的，这意味着它们都是自由空间并且可以被提交到物理内存，或者为将来使用而保留起来。任何自由状态地址在能够被使用前，必须首先被分配为保留状态或已提交状态。

当使用标准内存管理函数分配内存的时候，用户无法指定内存块位于哪个线程地址，或者不要位于哪个线程地址，而使用虚拟内存管理函数可以做到这一点。但这样做的理由是什么呢？考虑这样一种情况：程序需要一个内存块用做缓冲区，随着程序的运行，这个内存块可能随时需要扩展，最大可能扩展为 100 MB 大小，所以希望系统在分配其他内存块的时候不要使用这个内存块后面 100 MB 大小范围内的地址空间，这样，就可以随时将内存块扩大而不必移动它的位置。

除了这样一个主要的用途外，虚拟内存管理函数还提供转换虚拟地址空间页状态的能力，一个应用程序可以把内存的状态从已提交改变为保留，或把保护的属性从 `PAGE_READWRITE`（可读写）改变为 `PAGE_READONLY`（只读），从而防止对某段地址空间的写访问；应用程序也可以锁定一页内存，不让它被交换到磁盘中。

虚拟内存管理函数是一组名字以 `Virtual` 开头的函数，主要包括下面几种：

- `VirtualAlloc` 和 `VirtualFree`——进行地址空间的分配和释放工作。
- `VirtualLock` 和 `VirtualUnlock`——对内存页进行锁定和解锁。
- `VirtualQuery` 或 `VirtualQueryEx`——查询内存页的状态。
- `VirtualProtect` 或 `VirtualProtectEx`——改变内存页的保护属性。

1. 保留和释放地址空间

保留或提交一段地址空间，使用 `VirtualAlloc` 函数；释放或解除提交地址空间，则使用 `VirtualFree` 函数。先来看 `VirtualAlloc` 函数的使用方法：

invoke `VirtualAlloc, lpAddress, dwSize, flAllocationType, flProtect`

`lpAddress` 参数指定需要保留或提交的地址空间的位置，参数可以使用 `NULL` 值也可以指定一个具体的地址。`NULL` 值表示由函数自行在某个最方便的位置保留地址范围，非 `NULL` 值指定了一个准确的初始地址。如果函数返回 `NULL`，表示执行失败，否则返回一个指针，指向被保留地址范围的开始位置。

`dwSize` 参数表示函数应该分配的地址范围大小，它可以是 0 B~2 GB 的任意值，但系统会

自动把它进位到一个页面的整数倍大小。另外，虽然参数的最大值可以指定为 2 GB，但实际上能够被保留的最大值是该进程中最大的连续自由地址空间。

`flAllocationType` 参数用来决定如何分配地址，它可以是以下取值的组合：

- `MEM_COMMIT`——为指定地址空间提交物理内存。
- `MEM_RESERVE`——保留指定地址空间，不分配物理内存。
- `MEM_TOP_DOWN`——尽可能使用高端的地址空间。

`flProtect` 参数用来指定保护的类型，它可以是以下取值之一：

- `PAGE_READONLY`——为已提交物理内存的地址空间设定只读属性。
- `PAGE_READWRITE`——为已提交物理内存的地址空间设定可读写属性。
- `PAGE_EXECUTE`——为已提交物理内存的地址空间设定可执行属性。
- `PAGE_EXECUTE_READ`——为已提交物理内存的地址空间设定可读和可执行属性。
- `PAGE_EXECUTE_READWRITE`——为已提交物理内存的地址空间设定可读、可写和可执行属性。
- `PAGE_NOACCESS`——将保留的地址空间设定为不可存取模式。

`VirtualFree` 函数的使用语法是：

```
invoke    VirtualFree, lpAddress, dwSize, dwFreeType
```

`lpAddress` 和 `dwSize` 参数指定地址和地址空间的大小，`dwFreeType` 指定释放地址空间的方式，它可以是以下的数值：

- `MEM_DECOMMIT`——为一个已经提交物理内存的地址空间解除提交。
- `MEM_RELEASE`——释放保留的地址空间。

现在来看如何使用它们来保留地址空间和释放保留的地址空间。使用 `VirtualAlloc` 函数保留一个地址空间的分配方式使用 `MEM_RESERVE`，由于被保留的地址空间还没有提交给物理内存，是无法访问的，所以保护属性必须使用 `PAGE_NOACCESS` 标志，具体的语句是：

```
invoke    VirtualAlloc, NULL, 10485760, MEM_RESERVE, PAGE_NOACCESS
.if      eax
mov      lpAddress, eax
.endif
```

这一段代码导致系统保留一个 10 MB 大小的地址空间。当在一个进程中保留地址时，没有物理内存页被提交，也没有在页文件中为它保留空间，而只是阻止了其他内存分配函数对该段地址的请求而已，保留一个地址范围并不保证将来会有可用的物理内存来提交给这些地址。

保留地址的操作是很快的，保留一个小的地址范围和保留一个大范围的地址空间的速度差不多，因为在操作期间，并没有资源分配。

如果要释放保留的地址空间，可以使用 `MEM_RELEASE` 方式调用 `VirtualFree` 函数：

invoke	VirtualFree, lpAddress, 0, MEM_RELEASE
--------	--

lpAddress 就是上面调用 VirtualAlloc 返回的指针, dwSize 参数在这里必须为 0。当使用上面的 VirtualAlloc 函数保留了一段地址空间以后, 接下来还可以继续多次调用同样的函数提交这段地址空间中的不同页面, 所以到最后不同的页面可能处在不同的状态中(提交的和没有提交的)。如果用 VirtualFree 函数释放这个地址空间, 所有的页面必须处在相同的状态下(可以是全部提交的或全部没有提交的), 否则释放操作会失败。当不同页面的状态不同的时候, 最好首先将所有的已提交页面逐一解除提交, 最后再使用上面举例的方法释放整个地址空间。

有时候, 两次调用 VirtualAlloc 函数保留了两段连在一起的地址空间, 对于这种情况, 虽然两段地址空间实际上是连在一起的, 但也无法调用 VirtualFree 函数将它们一次释放, 必须调用两次 VirtualFree 函数将它们分别释放。

2. 使用保留的地址空间

要使用保留的地址, 首先必须提交物理内存给该地址。提交内存到地址与保留内存同样使用 VirtualAlloc 函数, 只是调用的方式使用 MEM_COMMIT 标志。在已经保留的地址段中, 内存可以按一页的大小被分次提交, 也可以一次提交所有的保留地址。

当内存被提交时, 可能全部被分配为物理内存页, 也可能一部分或全部被分配在页文件中, 直到它被访问。一旦内存页已提交, 系统就会像对待用其他函数分配的内存块一样来对待它们。

使用 VirtualAlloc 函数提交地址空间的方法是:

invoke	VirtualAlloc, lpAddress, 4096, MEM_COMMIT, PAGE_READWRITE
.if	eax
	mov lpMemory, eax
.endif	

这个语句将一个页面(4 096 字节)的保留地址提交到物理内存。在提交的时候, lpAddress 参数不能指定为 NULL, 而是要指定一个特定的地址来准确地指示被保留地址的哪一页会被提交。而且, 页的属性现在要指定是可以访问的, 不能再使用 PAGE_NOACCESS, 可以使用 PAGE_READWRITE 和 PAGE_READONLY 等属性。如果函数执行成功, 返回的是被提交地址中第一页的起始线程地址, 执行失败将返回 NULL。

提交内存的时候, 系统只能按页面的整数倍大小提交, 函数会自动按照 lpAddress 和 dwSize 指定的范围把与这个范围同属一个页面的地址全部提交, 所以当 lpAddress 指定的数值不是一个页的整数倍的时候, 返回的 lpMemory 就不会和指定的 lpAddress 相同, 而是被修改为页的边界地址。

如果要一次提交全部保留的地址空间, 那么可以把保留和提交的操作合并到同一次对 VirtualAlloc 函数的调用中:

invoke	VirtualAlloc, NULL, dwSize, MEM_RESERVE or MEM_COMMIT, PAGE_READWRITE
.if	eax
	mov lpMemory, eax
.endif	

这种方法与用 GlobalAlloc 函数直接分配一块内存没有多大的差别，惟一的好处就是可以自己指定分配的内存块地址。

如果想对已经提交的页面解除提交，让它们从提交状态返回到保留状态，可以使用 VirtualFree 函数，这时需要使用 MEM_DECOMMIT 参数：

invoke	VirtualFree, lpMemory, dwSize, MEM_DECOMMIT
--------	---

同样，函数操作的对象是整个页面，如果指定的内存范围不是整个页面，函数会自动将整个范围同属一个页面的地址全部解除提交。

3. 内存页的保护和锁定

除了用 VirtualAlloc 函数在提交内存的时候指定不同的保护方式外，也可以在以后用 VirtualProtect 函数来改变虚拟内存页的保护方式。比如，应用程序可以按 PAGE_READWRITE 来提交一个页并立即将数据写到该页中，然后马上使用 VirtualProtect 函数将该页的保护方式改为 PAGE_READONLY，这样可以有效地保护数据不被该进程中的任何线程重写。VirtualProtect 函数的用法是这样的：

invoke	VirtualProtect, lpAddress, dwSize, flNewProtect, lpflOldProtect
--------	---

flNewProtect 是新的保护方式，取值可以参考 VirtualAlloc 函数中的 flProtect 参数，lpflOldProtect 是指向一个双字的指针，函数会在这里返回原来的保护方式，如果不需要知道原来的方式，可以把这个参数设置为 NULL。

VirtualProtect 函数还可以用在什么地方呢？MSDN 中由 Randy Kath 书写的一篇文章“Managing Virtual Memory in Win32”中的例子很有代表性：

“一个用于缓冲数据的应用程序接收到一组大小变化的数据流，由于其他应用程序对 CPU 时间的竞争，数据流可能在某些时候超出进程的能力。为了防止这种现象发生，应用程序可以在开始时为一个缓冲区提交一些内存页，然后使用 PAGE_NOACCESS 保护来保护内存的顶端页，使得任何想要访问该内存的请求都会产生一个异常。应用程序也在该代码的外层代码中使用一个异常处理程序来处理访问冲突。”

“当处理能力不够的时候，缓冲区会满到这个受保护的顶端页，于是会产生一个访问冲突，这时应用程序就知道缓冲区已经到了其极限，该应用程序可以通过将页保护改变为 PAGE_READWRITE 来响应，允许该缓冲区接收任何附加的数据，并且继续不间断地执行。同时，应用程序加载另一个线程来减缓数据流，直到该缓冲区恢复到一个理想的操作范围。当情况恢复到正常，顶端的页又返回为 PAGE_NOACCESS 页，附加的线程也结束了。这样可以将页保护和异常处理程序结合起来提供独一无二的内存管理机会。”

另外，应用程序还可以使用 VirtualLock 和 VirtualUnlock 函数，它们的功能分别是将内存页锁定在物理内存中以及解除锁定。这两个函数的语法很简单：

invoke	VirtualLock, lpAddress, dwSize
invoke	VirtualUnlock, lpAddress, dwSize

“锁定”的意思是要求系统总是将指定的内存页保留在物理内存中，不许将它交换到磁盘

页文件中。如果程序中有些内存被频繁使用，将它们保留在物理内存可以提高访问的速度。由于锁定太多的页面会导致其他页面被频繁交换到页文件中，所以 Windows 限制每个进程能同时锁定的页数不能超过 30 个。只有已经被提交的内存页才能被锁定，对一个保留的地址进行锁定操作是不能成功的。

10.1.6 其他内存管理函数

Win32 中还有其他的一些内存管理函数，可以用来完成一些辅助的功能，如内存填充、移动以及测试函数等。

1. 填充和移动内存

填充和移动内存本来就可以用几句简单的代码实现，如下面的代码可以将从 `szSource` 开始的 `dwSize` 大小的内存块移动到 `szDest` 处：

```
mov     esi,offset szSource
mov     edi,offset szDest
mov     ecx,dwSize
cld
rep     movsb
```

而下面的代码可以将 `szDest` 处的 `dwSize` 字节填充为 0：

```
xor     eax,eax
mov     edi,offset szDest
mov     ecx,dwSize
cld
rep     stosb
```

如果把 `xor eax, eax` 换成 `mov al, xx`，那么上面代码的功能就是将这块内存填充为 `xx`。

虽然填充和移动的功能这么简单，但 Win32 中还是有对应的 API 函数：

```
invoke  RtlMoveMemory,offset szDest,offset szSource,dwSize ;移动内存
invoke  RtlFillMemory,offset szDest,dwSize,dbFill ;以 dbFill 填充内存块
invoke  RtlZeroMemory,offset szDest,dwSize ;以 0 填充内存块
```

可以看到，使用这些函数时，仅传递参数和调用的开销就远远超过了前面举例的两段代码，但是由于使用它们的可读性比较好，所以在具体的使用中要有所取舍。如果执行速度比较重要，比如，在一个循环中使用，同样的代码要被使用很多遍，还是应该使用嵌入的几句汇编代码；如果为了让程序看上去简洁一些，那就不妨使用这几个 API 函数。

2. 内存状态测试

有时候在访问一块内存之前，可能想知道这块内存的属性究竟是什么，是可写的？可读的？还是可执行的？这些功能可以用测试函数来完成：

```
invoke  IsBadCodePtr, lpMemory
invoke  IsBadReadPtr, lpMemory, dwSize
invoke  IsBadWritePtr, lpMemory, dwSize
invoke  IsBadStringPtr, lpMemory, dwSize
```

这些函数的功能如下：

- IsBadCodePtr 函数测试某个指针指向的单个字节是否可读，如果可读则返回 0，否则返回非 0 值。
- IsBadReadPtr 函数测试某段内存是否可读，如果这段内存的所有字节都是可读的，则返回 0，如果中间包含有不可读的字节则返回非 0 值。
- IsBadWritePtr 函数测试某段内存是否可写，如果这段内存的所有字节都是可写的，则返回 0，如果中间包含有不可写的字节则返回非 0 值。
- IsBadStringPtr 函数测试的同样是可读性，lpMemory 参数指向一个以 0 结尾的字符串，字符串的最大长度为 dwSize，如果整个字符串包含结尾的一个 0 都是可读的，则函数返回 0，否则返回非 0 值。缓冲区中剩余的字节则不予测试。

10.2 文件操作

10.2.1 Windows 的文件 I/O

在 DOS 操作系统下，最早的文件操作方法是使用 FCB（文件控制块），FCB 是一个数据结构，为了存取一个文件，必须建立一个 FCB 并在其中填写好驱动器名、文件名和要读写的记录号等，然后调用 int 21h 中对应的功能。使用 FCB 方式的缺点很多，如每次只能按记录为单位读取数据，无法随意指定数据块大小，无法直接指定一个全路径的文件名，文件的操作位置不会自动调整，每次操作都必须指定记录号等，归纳起来就是功能简单，操作复杂。

于是在 2.0 以上的 DOS 版本中，开始使用更方便的文件句柄方式，这种方式不再需要文件控制块，程序指定一个包含全路径的文件名后，就可以要求操作系统打开这个文件并返回一个文件句柄，以后就可以用这个句柄来读写文件，直到关闭文件为止。操作系统在内部为每个文件句柄维护一个读写指针。读写指针总是指向文件下次要存取的位置，每次对文件的读写操作完成以后，读写指针会自动调整到本次操作的最后一个字节后面的位置，这样顺序读写文件就不必每次重新指定位置。读写指针可以被移动到文件的任意位置，以便满足随机存取的要求。

Windows 操作系统中，文件操作沿用了这种句柄方式，保留了文件句柄和读写指针等概念，同时又根据 Windows 操作系统的新特征对文件 I/O 进行了很多的扩展，下面列出了 Win32 中文件函数经过扩展的一些功能：

- 文件函数的操作对象有了很大的扩展，除了普通的文件，对串口、磁盘设备、网络文件、控制台和目录等的操作都可以使用文件函数来完成。
- 支持异步文件操作，文件函数可以不必等待到操作完成才能返回。
- Windows 是多用户的操作系统，可能发生多个程序同时对文件操作的现象，文件函数中增强了对共享和锁定的支持。
- 文件操作函数和内存映射文件函数配合可以实现将文件当做内存的一部分来存取的功能。

- 增加了拷贝文件和移动文件等函数来实现常用的功能。

另外，在文件的命名中有长短文件名之分，众所周知，DOS 操作系统使用 8.3 结构的文件命名方式，在这种命名方式下，用文件名来简单地说明文件的用途显得比较困难，因为仅用 8 个字符是表达不了什么复杂的含义。

而在长文件名系统中，文件名的长度可以长达 255 个字符，这样在文件名中就可以清晰地表达出文件的用途，长文件名在磁盘的目录区中占用了多个连续的目录项，其中的一个目录项用做 8.3 结构的短文件名，其他的目录项存放其他名字字符。在 8.3 文件名中不合法的一些字符，如小数点与空格等在长文件名中都可以使用，只有 / \ : * ? " < > | 等 9 个字符不能用于长文件名。

长文件名需要文件系统的支持，从 DOS 到 Windows，使用过的有 FAT，VFAT，FAT32，NTFS 与 HPFS 等多种文件系统，在这些文件系统中，只有 FAT 系统不支持长文件名。

各种操作系统对文件系统的支持是不同的。Windows 3.x 和 DOS 操作系统一直使用的是文件分配表（FAT）系统；Windows 95 开始使用扩展 FAT 文件系统（VFAT），FAT 系统和 VFAT 系统都是 16 位的文件系统，也称为 FAT16。Windows NT 在支持 FAT16 的同时，还支持两种 32 位的文件系统：NT 文件系统（NTFS）和高性能文件系统（HPFS），NTFS 支持文件的安全性，能够指定谁能访问某一文件或目录和对它做什么操作。Windows 系列操作系统对文件系统的支持如表 10.2 所示：

表 10.2 Windows 系列操作系统对文件系统的支持

操作系统	支持 FAT16	支持 FAT32	支持 NTFS	支持 HPFS
Windows 95	√	×	×	×
Windows 98	√	√	×	×
Windows NT	√	×	√	√
Windows 2000/XP	√	√	√	√

那么在 Win32 的文件操作函数中，如何处理长、短文件名，又如何处理不同的文件系统呢？答案很简单：就是不要去考虑它们，不管要操作的文件名是长是短，不管文件位于什么样的文件系统中，只要指定了正确的文件名，文件操作函数就能正确地处理它。

10.2.2 创建和读写文件

在开始讨论文件 I/O 的函数之前，先来看一个例子，这是一个将 UNIX 文件格式的文本转换到 PC 格式文本的小程序 FormatText，由于 UNIX 系统保存的文本文件以 0ah 作为一行的结束而不是使用 0dh+0ah，造成这种格式的文本在 Windows 的记事本上无法看到正常的换行，所有的内容将被显示在同一行内。本程序将读取 UNIX 格式的文本文件，转换到 PC 格式并保存到一个新的文本文件中。文件的源程序可以在所附光盘的 Chapter10\FormatText 目录中找到。汇编源文件 FormatText.asm 的内容如下：

```
.386
.model flat, stdcall
option casemap :none
```



哭

[illegible]



[illegible]

资源文件 FormatText.rc 如下:

```
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#include <resource.h>
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
#define ICO_MAIN 1000
#define DLG_MAIN 100
#define IDC_FILE 101
#define IDC_BROWSE 102
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAINICON "Main.ico"
//>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
DLG_MAIN DIALOG 84, 79, 201, 41
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "UNIX Text file -> PC Text file"
FONT 9, "宋体"
{
LTEXT "文件名", -1, 7, 8, 25, 8
EDITTEXT IDC_FILE, 35, 5, 160, 12, ES_AUTOHSCROLL | WS_BORDER | WS_TABSTOP
```

程序在_ProcFile 子程序中打开输入的 UNIX 文本文件，并创建用于输出的新文件，然后多次将文件数据读入到缓冲区中并调用_FormatText 子程序处理字节流，_FormatText 子程序找出每行的结尾字符 0ah 并在前面加上 0dh 后存盘。

1. 打开和关闭文件

打开文件使用的函数是 `CreateFile`，读者一定认为是搞错了，这是创建文件，打开文件不是 `OpenFile` 吗？没有错！在 Win32 中不能使用 `OpenFile` 函数（Win16 中倒是有 `OpenFile` 函数），不管是打开文件还是创建文件使用的都是 `CreateFile` 函数。`CreateFile` 函数的参数又多又复杂，读者千万不要被下面的内容吓住了。

invoke	CreateFile, lpFileName, dwDesiredAccess, \dwShareMode, lpSecurityAttributes, dwCreationDisposition, \dwFlagsAndAttributes, hTemplateFile
--------	--

lpFileName 指向存放有文件名的缓冲区，文件名是一个以 0 结尾的字符串，字符串的最大长度为 MAX_PATH（这个值在 Windows.inc 文件中定义为 260），Win32 可以用文件函数处理多种对象，所以 CreateFile 函数可以打开多种对象，包括：

- CreateFile 以不同格式的文件名来区分操作的对象，普通的“驱动器:\路径\文件名”之类的格式指的就是普通的文件；当指定“COM1”为文件名的时候，操作的对象是第一个串口；如果文件名是“\\.\mailslot\filename”格式，那么要操作的就是邮件槽了；要打开的文件也可以是网络上其他主机中的文件，这时可以用“\\服务器名\共享目录名\文件名”方式来指定文件名。在本章中，只讨论对普通文件的操作。

dwDesiredAccess 参数是存取方式,通过这个参数可以指定要对打开的文件进行何种操作,指定 GENERIC_READ 标志表示需要读取文件数据,指定 GENERIC_WRITE 标志表示需要向文件写数据,如果要对一个文件进行读写,需要同时指定这两个标志。

dwShareMode 参数是共享属性,表明文件被打开后是否允许其他进程以某种方式再次打开文件,它可以是一些取值的组合:

- 0——不允许文件再被打开。
- FILE_SHARE_DELETE——许其他进程同时对文件进行删除。
- FILE_SHARE_READ——允许其他进程同时以读方式打开文件。
- FILE_SHARE_WRITE——允许其他进程同时以写方式打开文件。

lpSecurityAttributes 参数为安全属性,通过这个参数可以指定返回的文件句柄是否可以被子进程继承,如果参数设置为 NULL,则表明无法被继承,否则需要将参数指向一个 SECURITY_ATTRIBUTES 结构,该结构的定义为:

SECURITY_ATTRIBUTES STRUCT				
nLength	DWORD	?		;本结构的长度
lpSecurityDescriptor	DWORD	?		
bInheritHandle	DWORD	?		;是否允许继承
SECURITY_ATTRIBUTES ENDS				

nLength 字段需要设置为结构的长度,将 bInheritHandle 字段设置为 TRUE 就可以使句柄能够被子进程继承。

dwCreationDisposition 参数用来设置文件已经存在或不存在时系统采取的动作,在这里指定不同的标志就可以决定函数执行的功能究竟是创建文件还是打开文件,参数可能的取值为:

- CREATE_NEW——创建新文件,如果文件已经存在函数会返回失败。
- CREATE_ALWAYS——创建新文件,如果文件已经存在则清除原文件。
- OPEN_EXISTING——打开存在的文件,当文件不存在时函数会返回失败。
- OPEN_ALWAYS——如果文件已经存在,则打开,不存在则创建新文件。
- TRUNCATE_EXISTING——打开文件并将文件截断为零,当文件不存在时返回失败。

dwFlagsAndAttributes 参数用来指定新建文件的属性,文件属性可以是下面这些值的组合:

- FILE_ATTRIBUTE_NORMAL——普通文件,设置这个属性时其他属性都不会生效。
- FILE_ATTRIBUTE_ARCHIVE——设置归档属性。
- FILE_ATTRIBUTE_HIDDEN——设置隐藏属性。
- FILE_ATTRIBUTE_READONLY——设置只读属性。
- FILE_ATTRIBUTE_SYSTEM——设置系统属性。
- FILE_ATTRIBUTE_TEMPORARY——临时文件,系统会尽量把所有的文件内容保持在内

存中以加快存取速度，程序在不再使用文件的时候需尽快将它删除。

此外该参数还可同时指定对文件操作的方式，常用的方式有：

- FILE_FLAG_WRITE_THROUGH——使用 WriteThrough 模式，系统不会对文件使用缓存，文件的改变马上会被写入到磁盘中。
- FILE_FLAG_OVERLAPPED——使用异步文件操作模式。
- FILE_FLAG_DELETE_ON_CLOSE——文件被关闭后立即被系统自动删除。
- FILE_FLAG_RANDOM_ACCESS——对文件进行随机读写操作（操作系统对该文件的缓存进行优化）。

hTemplateFile 指定了一个文件模板的句柄，该文件模板的所有属性都会被复制到当前创建的文件中。Windows 95 不支持本参数，为了保持程序的兼容性，建议在参数中使用 NULL。

当打开或创建文件成功的时候，函数返回一个文件句柄，失败的话，函数的返回值是 INVALID_HANDLE_VALUE，注意：这个值被定义为-1 而不是 NULL。如果想再详细地了解失败的原因，可以继续调用 GetLastError 函数。

用不同参数的组合调用 CreateFile 可以完成不同的功能，比如在例子中打开输入的文本文件的时候，使用的是下面的代码：

```

invoke    CreateFile,addr szFileName,GENERIC_READ,FILE_SHARE_READ,\
          0,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0
.if       eax == INVALID_HANDLE_VALUE
          ;输出出错信息
.endif
mov       @hFile,eax

```

为什么要这样使用呢？因为如果文件不存在的话，就没有必要继续处理，所以要使用 OPEN_EXISTING 标志，而且程序仅需要读取文件的内容，不需要写入数据，所以存取方式使用 GENERIC_READ 就可以了，为了让其他程序能同时使用这个文件，共享方式要指定为 FILE_SHARE_READ。

而创建用于输出的文件时就不同了，这次使用的代码如下：

```

invoke    CreateFile,addr @szNewFile,GENERIC_WRITE,FILE_SHARE_READ,\
          0,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,0
.if       eax == INVALID_HANDLE_VALUE
          ;输出出错信息
.endif
mov       @hFileNew,eax

```

因为要写文件，所以存取方式使用 GENERIC_WRITE，打开方式使用 OPEN_ALWAYS，告诉函数如果记录文件已经存在，则打开它并将它的内容清除，不存在的话则新建一个文件。

与打开或创建文件相比，关闭文件就简单得多了，关闭文件使用 CloseHandle 函数：

```

invoke    CloseHandle,hFile

```

2. 移动文件指针

系统为每个打开的文件维护一个文件指针，指定在文件中下一个读操作或写操作在什么位置进行，随着数据的读出或写入，文件指针随之移动。当文件刚被打开的时候，文件指针处于文件的头部。

顺序读取或写入文件数据的时候，由于文件指针总是被调整到上一次操作以后的地方，所以循环进行读取或写入就能将全部的文件数据处理完毕。但使用中常常需要随机读取文件内容，这些就需要先调整文件指针，再进行读写操作。

调整文件指针使用 `SetFilePointer` 函数：

invoke	<code>SetFilePointer, hFile, lDistanceToMove, \</code> <code>lpDistanceToMoveHigh, dwMoveMethod</code>
--------	---

`hFile` 是用 `CreateFile` 函数返回的文件句柄。

`lDistanceToMove` 指定要移动的距离。`lpDistanceToMoveHigh` 是一个指针，指向一个 32 位的变量，变量存放有移动距离的高 32 位，它和 `lDistanceToMove` 中的 32 位一起组成一个 64 位的距离，使用 64 位距离的原因是某些平台上的 Windows 支持 64 位的文件长度，但在运行于 80x86 平台上的 Windows 版本中，文件长度不会超过 4 GB，只需 32 位就能够覆盖全部的文件长度，所以 `lpDistanceToMoveHigh` 参数一般设置为 `NULL`（其他的文件操作函数也使用两个参数来指定 64 位的文件长度，同样也可以只使用低 32 位）。当移动距离是正值的时候，文件指针向文件尾部移动；如果移动距离是负值，那么文件指针向文件头部方向移动。

`dwMoveMethod` 指定了移动的模式，也就是指明从什么地方开始移动，它可以是以下的取值：

- `FILE_BEGIN`——不管文件指针当前位于什么地方，总是从文件头部开始移动，这时的位置参数相当于指定了一个绝对位置。
- `FILE_CURRENT`——从当前的文件指针处开始移动，这时的位置参数相当于指定了一个相对位置。
- `FILE_END`——从文件尾开始移动，如果要从文件尾往回移动一段位置，那么位置参数指定的就应该是负值。

Win32 文件操作函数可以支持很多对象，有些对象并不支持文件指针，对它们就不能使用 `SetFilePointer` 函数，能想象对打开的串口进行移动指针操作是什么意思吗？

函数的返回值根据 `lpDistanceToMoveHigh` 参数的取值不同而不同，由于这个参数一般设置为 `NULL`，这里仅讨论参数为 `NULL` 时的情况。这时如果函数执行失败，返回值是 -1，否则函数返回新的文件指针位置。

既然文件指针可以设置，那么如何获取当前的文件指针呢？实际上并没有一个专用的函数可以完成这个功能（并没有 `GetFilePointer` 函数），但是既然 `SetFilePointer` 函数的返回值是新的文件指针位置，那么可以巧妙地使用该函数来获得当前的文件指针：

invoke	<code>SetFilePointer, hFile, 0, NULL, FILE_CURRENT</code>
--------	---

从当前的文件指针处移动 0 字节，文件指针并没有真正移动，但是返回值就是当前的文件指针了！

文件指针也可以移动到文件所有数据的后面，比如，现在文件的长度是 100 B，但还是可以成功地把文件指针移到 1 000 B 的位置，这样的操作有什么用途呢？用途是可以将文件扩展到需要的长度，可以接着用 WriteFile 写入数据，系统会将文件从 100 B 扩展到 1 000 B 后再从 1 000 B 处写入数据。

使用 SetEndOfFile 函数也可以扩展文件长度，SetEndOfFile 总是将文件的长度调整到当前的文件指针指向的长度，所以这个函数还有截断文件的功能，当文件指针位于文件中间的时候，函数将文件指针后面的内容截断，当文件指针位于文件尾以后位置的时候，函数将文件长度扩展。SetEndOfFile 函数的用法是这样的：

invoke	SetEndOfFile, hFile
--------	---------------------

当文件被扩展的时候，被扩展部分的内容是不确定的（不过这是 MSDN 说的，试验的结果好像这部分内容总是 0）。

SetEndOfFile 函数总是要和 SetFilePointer 函数配合使用，比如，要写一个杀毒程序，需要将附在带毒软件尾部的病毒去掉，就可以先用 SetFilePointer 函数将文件指针移到文件原来长度的地方，再用 SetEndOfFile 函数将文件截断就可以了。另外，要建立一个内容为空的文件（并不是指文件长度为 0），可以在创建文件后，马上将文件指针移到需要的地方，再用 SetEndOfFile 函数扩展文件就可以了。网络蚂蚁在下载前总是创建一个和目标文件同样尺寸的空文件，然后再逐步下载其中的数据，用这种方法就可以完成创建空文件的功能。

3. 读写文件

读写文件可以使用 ReadFile/WriteFile 函数，这两个函数读写的方式可以是同步的也可以是异步的；也可以使用 ReadFileEx/WriteFileEx 函数，这两个函数只用于异步读写文件。

读文件函数 ReadFile 的使用方法是：

invoke	ReadFile, hFile, lpBuffer, nNumberOfBytesToRead, \
	lpNumberOfBytesRead, lpOverlapped

其中 hFile 是文件句柄，用来指明要读取的文件；lpBuffer 指向一个缓冲区，函数会将读出的数据传送到这里；nNumberOfBytesToRead 参数指定需要读入的字节数，由于函数并不能总是读到用户要求的字节数（原因是多种多样的，比如，遇到了文件尾），所以下面一个参数 lpNumberOfBytesRead 指向一个 dword 类型的变量，函数将在这里返回实际读入的字节数；lpOverlapped 参数指向一个 OVERLAPPED 结构，供函数在异步读取文件时使用，在同步读写中这个参数设置为 NULL。由于 Windows 95 的 ReadFile 函数并不支持对文件的异步读写，所以在 Windows 95 中使用时，这个参数总是设置为 NULL。

如果读取文件失败，则函数返回 0，成功则函数返回非 0 值。当函数返回非 0 值而 lpNumberOfBytesRead 中返回的已读取字节数却是 0 时，表示已经读到了文件尾。在例子中读文件使用：

invoke	ReadFile, @hFile, addr @szReadBuffer, \
	sizeof @szReadBuffer, addr @dwBytesRead, 0

这样读出的字节数保存在@dwBytesRead 变量中，然后根据@dwBytesRead 中的计数循环处理文件内容。

向文件写数据的函数 WriteFile 的使用方法与 ReadFile 函数类似：

invoke	WriteFile, hFile, lpBuffer, nNumberOfBytesToWrite, \
	lpNumberOfBytesWritten, lpOverlapped

同样，hFile 参数是文件句柄；lpBuffer 指向一个缓冲区，缓冲区中包含有要写入文件的数据；nNumberOfBytesToWrite 参数指定需要写入的字节数，函数在 lpNumberOfBytesRead 指出的 dword 类型变量中返回成功写入的字节数。

当用 WriteFile 写文件的时候，写入的数据可能被 Windows 暂时保存在内部的高速缓存中，等合适的时候再一并写入磁盘；如果 WriteFile 函数写的是串口或者一个管道，大块数据同样是暂时保留在缓冲区中等待逐步发送出去。虽然这些数据一般不会丢失，但并不能保证它们总是不会丢失的，比如，在文件关闭之前计算机断电了，或者数据发送之前对端断开了串口连接等情况。

如果一定要保证数据正确地写入了或者传输了，可以强制使用 FlushFileBuffers 函数来清空数据缓冲区，FlushFileBuffers 函数的参数仅是一个文件句柄：

invoke	FlushFileBuffers, hFile
--------	-------------------------

如果 hFile 代表一个文件，函数会将所有缓冲区的数据马上写入文件；如果 hFile 代表一个串口，那么函数发送缓冲区中所有的数据；如果 hFile 代表一个管道的话，函数一直等待直到管道的另一方读取全部数据。总之，如果这个函数执行成功的话，就能够保证所有的数据已经被传送。如果函数执行成功，返回值是非 0 值，执行失败则函数返回 0。

4. 文件的共享

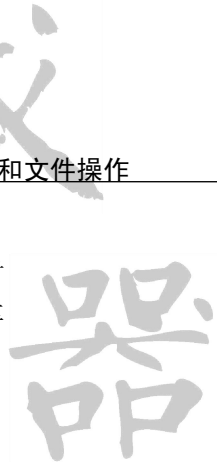
如果对文件数据的一致性要求比较高，为了防止程序在写入的过程中其他进程刚好在读取写入区域的内容，可以对已打开文件的某个部分进行加锁，加锁后可以防止其他进程对该区域进行读取或写入的操作。加锁和解锁使用 LockFile 和 UnlockFile 函数，读者也可以使用 LockFileEx 和 UnlockFileEx 函数，这两个函数可以以异步方式执行。

LockFile 函数和 UnlockFile 函数的使用方法是：

invoke	LockFile, hFile, dwFileOffsetLow, dwFileOffsetHigh, \
	nNumberOfBytesToLockLow, nNumberOfBytesToLockHigh
invoke	UnlockFile, hFile, dwFileOffsetLow, dwFileOffsetHigh, \
	nNumberOfBytesToLockLow, nNumberOfBytesToLockHigh

dwFileOffsetLow 和 dwFileOffsetHigh 参数组合起来指定了加锁区域的开始位置，dwNumberOfBytesToLockLow 和 dwNumberOfBytesToLockHigh 参数组合则指定加锁区域的大小，这两组参数都指定了一个 64 位的值，在 Windows 中，可以只使用低 32 位。

文件锁是排他性的，不能对一个区域重复加锁，两个不同的加锁区域也不能重叠；在对文件加锁和解锁的时候操作的区域也必须一一对应，对一个区域加锁后不能分几次对区域中的不同部分解锁，而必须一次全部解锁。



当程序读取一个文件的时候，如果文件有可能被其他进程加锁的话，那么在读取失败的时候就必须调用 GetLastError 函数来获取失败的原因，如果失败的原因是因为锁定造成的共享错误的话，那么可以等待一段时间后继续读取。

10.2.3 查找文件

在 DOS 中可以使用 int 21h 中断的 4eh 和 4fh 功能查找一个目录中的指定文件，Win32 中也可以进行类似的操作，方法是用两个函数分别实现查找第一个文件和继续查找文件的功能。

当要开始查找文件的时候，首先使用 FindFirstFile 函数，如果函数执行成功，返回一个句柄 hFindFile 来对应这个寻找操作，接下来可以利用这个句柄循环调用 FindNextFile 函数继续查找其他文件，一直到 FindNextFile 函数返回失败为止，最后必须关闭 hFindFile 句柄。使用这几个函数查找文件的代码一般使用下面的结构：

```

invoke    FindFirstFile, lpFindFile, lpFindFileData
.if      eax != INVALID_HANDLE_VALUE
mov      hFindFile, eax
.repeat
        ;处理本次找到的文件
        invoke    FindNextFile, hFindFile, addr lpFindFileData
.until   eax == FALSE
        invoke    FindClose, hFindFile
.endif

```

如果 FindFirstFile 函数执行失败，则下面的循环就不必执行了，如果 FindFirstFile 执行成功，那么程序保存返回的 hFindFile 并开始一个 .repeat 循环，使用 .repeat 语句而不是 .while 的原因是 .repeat 构成的循环先执行循环体内的内容，这样第一次循环时首先处理 FindFirstFile 找到的文件，然后再根据 FindNextFile 执行的结果决定是否继续循环。在结束循环后，需要用 FindClose 函数将查找句柄关闭。

FindFirstFile 的参数 lpFindFile 指向一个字符串，代表要寻找的文件名，如果文件名中不包含路径，那么在当前目录中查找文件，包含路径的话将在指定路径中查找。在文件名中可以用 “*” 或 “?” 通配符指定查找特定的文件。下面是文件名格式的几个例子：

c:\Windows*.*	;在 c:\Windows 目录中查找所有文件
c:\Windows\System32*.dll	;在 c:\Windows\System32 目录中查找所有 dll 文件
c:\Windows\System.ini	;在 c:\Windows 目录中查找 System.ini 文件
c:\Windows\a???.*	;在 c:\Windows 目录中查找所有以 a 开头的文件名
	;长度为 4 个字符的文件
Test.dat	;在当前目录查找 Test.dat 文件
.	;在当前目录查找所有文件

FindFirstFile 函数和 FindNextFile 函数中的 lpFindFileData 参数则指向一个缓冲区，函数会在缓冲区中返回一个 WIN32_FIND_DATA 结构，结构中包括了 Windows 查找过程中临时使用的数据和找到的文件名与文件属性等数据。该结构的定义如下：

```

WIN32_FIND_DATA STRUCT
    dwFileAttributes    DWORD        ?           ;文件属性
    ftCreationTime       FILETIME <>         ;文件的创建日期

```

哭

[illegible]

哭

[illegible]

例子文件的资源文件 FindFile.rc 如下:

[illegible]

例子程序查找指定目录，以及所有下层子目录中的所有文件，然后累计文件的长度，以此计算目录中所有文件的总长度，当单击对话框中的“浏览”按钮的时候，程序调用附录 C 中介绍的“浏览目录”通用对话框获取需要查找的目录，源程序中用到了附录 C 中的_BrowseFolder 子程序。

在 FindFile 子程序中，使用了前面推荐的循环格式：

注意：这个子程序是递归调用的，这样不管下层子目录的深度有多少，程序都不需要去考虑它。如果不使用递归调用的方法，那么程序必须将找到的子目录先保存起来，等处理完当前目录的所有文件后再继续处理保存的目录。

由于 WIN32_FIND_DATA 结构中的 cFileName 字段不包括路径，所以在找到一个文件后，程序用 lstrcpy 和 lstrcat 函数将保存的路径和文件名连接到一起组成一个全路径的文件名，然后，程序测试 dwFileAttributes 中的属性。如果找到的是目录，则递归调用 _FindFile 子程序寻找下一层目录；如果不是目录，则调用 _ProcessFile 子程序处理找到的文件。

在 _ProcessFile 子程序中打开文件并获取文件长度进行累计。如果要把这个全盘搜索程序用在别的地方（比如，编写杀毒程序的时候），只要将 _ProcessFile 子程序替换成对文件进行病毒检查的子程序就可以了。

当找到一个目录以后，还要判断目录名是否是“.”和“..”，如果是这两个名称则不进行处理，因为“.”目录代表本目录，“..”目录代表上一层目录，如果对它们进行递归查找，那么这个循环可就真的出不来了。

.repeat 循环中多了一个对 dwOption 是否是 F_STOP 的测试，这是为了随时终止查找过程。因为找完整个磁盘可能需要很长的时间，让用户有机会终止查找过程是程序人性化的表现。

对这个程序进行简单的修改，读者就可以将它用在任何需要全盘查找文件的地方，惟一需要注意的地方是：如果要查找的是“*.exe”之类的文件而不是“*.*”的话，在输入查找文件名的时候不能使用“*.exe”，因为这样会漏过子目录，正确的做法是使用“*.*”当做要查找的文件名，并在处理找到的文件时再对文件名进行判别，对扩展名不是.exe的文件直接忽略掉就可以了。

10.2.4 文件属性

一个文件有多种相关属性，如文件的类型、长度、日期，文件是只读的或隐含的等，Win32 中有一些函数可以对这些属性进行操作，在本节中将对此进行讨论。

1. 获取文件类型

Win32 的文件函数可以操作多种对象，如果要知道一个文件句柄究竟对应什么对象，可以使用 GetFileType 函数：

invoke	GetFileType, hFile
--------	--------------------

函数的返回值可能是下面的一种：FILE_TYPE_UNKNOWN, FILE_TYPE_DISK, FILE_TYPE_CHAR 或者 FILE_TYPE_PIPE，分别代表文件类型未知，文件句柄对应磁盘文件，文件句柄对应字符设备（如控制台或并行口等）和对应管道这 4 种情况。

2. 获取文件长度

如果需要得知文件当前的长度，可以使用 GetFileSize 函数：

```
invoke GetFileSize, hFile, lpFileSizeHigh
```

lpFileSizeHigh 指向一个用来接收高 32 位长度的变量，一般设置为 NULL，长度的低 32 位在返回值中返回，用 GetFileSize 函数得到的文件长度是“当前”长度，也就是说，假如现在文件长度是 100 字节，我们又在文件尾写了 100 字节，那么再一次调用函数得到的文件长度就是 200 字节。

3. 获取和修改文件日期

如果要获取文件的日期，可以使用 GetFileTime 函数：

```
invoke GetFileTime, hFile, lpCreationTime, lpLastAccessTime, lpLastWriteTime
```

其中，最后的 3 个参数指向 3 个缓冲区，函数会将创建日期、最后存取日期与最后写入日期分别返回到 3 个缓冲区中，如果不需要某个日期数据，可以把对应的输入参数设置为 NULL。返回的日期数据是个 FILETIME 结构，结构的定义为：

```
FILETIME STRUCT
    dwLowDateTime    DWORD    ? ;文件日期低 32 位
    dwHighDateTime   DWORD    ? ;文件日期高 32 位
FILETIME ENDS
```

可以看到，在这个结构中无法直接得到日期的年、月、日、时、分与秒等数据，为了将它转换成我们熟悉的 SYSTEMTIME 结构，需要调用 FileTimeToSystemTime 函数进行转换：

```
invoke FileTimeToSystemTime, lpFileTime, lpSystemTime
```

与 GetFileTime 函数相对应，也可以使用 SetFileTime 函数把文件日期设置成希望的日期，同样，输入的参数指向 3 个 FILETIME 结构，结构中预先填写好期望的日期参数，不需要修改的日期属性可以在参数中使用 NULL。函数用法如下：

```
invoke SetFileTime, hFile, lpCreationTime, lpLastAccessTime, lpLastWriteTime
```

同样，填写 FILETIME 结构的时候，最简便的方法是使用 SystemTimeToFileTime 函数将预先填写好的 SYSTEMTIME 结构转换为 FILETIME 结构：

```
invoke SystemTimeToFileTime, lpSystemTime, lpFileTime
```

4. 获取和修改文件属性

在创建文件的时候，可以在 CreateFile 的 dwFlagsAndAttributes 参数中指定文件属性，除了用这种方法设置文件属性外，也可以在以后使用 SetFileAttributes 参数修改文件属性：

```
invoke SetFileAttributes, lpFileName, dwFileAttributes
```

调用 SetFileAttributes 函数的时候不需要打开文件，只需要指定全路径的文件名就可以了，dwFileAttributes 参数的定义与 CreateFile 函数中的 dwFileAttributes 参数是一样的。

要获取文件的只读、隐含与系统等属性的话，可以使用 GetFileAttributes 函数：

```
invoke GetFileAttributes, lpFileName
```

如果函数执行失败，返回值是-1，否则返回文件的属性。

10.2.5 其他文件操作

在 DOS 操作系统中，并没有一个中断功能对应拷贝文件和移动文件功能，如果要实现这些功能，需要自己编写代码读取一个文件的内容并写入另一个文件，而在 Win32 中实现这些功能很简单，因为系统提供了对应的 API 函数。

在这一小节中，将讨论这些函数。

1. 拷贝文件

拷贝文件使用 CopyFile 或 CopyFileEx 函数，CopyFile 函数的用法如下：

```
invoke CopyFile, lpExistingFileName, lpNewFileName, bFailIfExists
```

函数将 lpExistingFileName 指定的文件拷贝为 lpNewFileName 指定的文件，最后一个参数 bFailIfExists 指定目标文件存在时的动作，如果指定为 TRUE，则拷贝失败；指定为 FALSE，则函数继续拷贝并覆盖掉原来的文件，拷贝出来的新文件的属性（如只读或隐含等属性）与原来文件的属性一样。

CopyFileEx 函数可以完成同样的功能，只不过函数可以指定一个回调函数，在拷贝的过程中，函数在每拷贝完一部分数据以后会调用回调函数，在回调函数中程序可以指定继续拷贝还是停止，或者显示一个进度条来指示拷贝的进度，所以 CopyFileEx 函数只在拷贝特大型文件的时候比较有用。

2. 移动文件

移动文件使用 MoveFile 或 MoveFileEx 函数。MoveFile 函数的用法如下：

```
invoke MoveFile, lpExistingFileName, lpNewFileName
```

函数将 lpExistingFileName 指定的文件移动到 lpNewFileName 指定的目标位置，文件名可以同时改变或者不改变，目标文件必须不存在，否则函数调用失败。当函数执行成功的时候返回非 0 值，否则函数返回 0。

MoveFile 函数可以移动文件或者目录。当移动一个文件的时候，源文件和目标文件既可以处于相同的驱动器上也可以处于不同的驱动器上，函数会自动决定是否需要进行拷贝工作（如果处在相同的驱动器中，函数仅移动了一下目录项，并不进行数据拷贝）；当移动的对象是一个目录时，那么目标目录和源目录必须处在同一个驱动器中，函数会将目录中的所有文件包括所有的下层子目录一并移动到新的位置上。

与 MoveFile 函数对应，系统中也有一个 MoveFileEx 函数，不过不要受 CopyFileEx 函数的影响而认为 MoveFileEx 函数只是提供了一个回调函数，事实上完全不是这回事，MoveFileEx 函数是 MoveFile 函数的扩展。它的使用方法是：

```
invoke MoveFileEx, lpExistingFileName, lpNewFileName, dwFlags
```

MoveFileEx 函数增加了一个 dwFlags 参数来进行移动控制，参数可以是下面取值的组合：

- MOVEFILE_COPY_ALLOWED —— 允许拷贝数据，如果不指定这个标志，则移动文件的

时候不能将文件移动到不同的驱动器上（当然即使指定了这个标志，将目录移动到不同驱动器上也是不支持的），本标志不能与 MOVEFILE_DELAY_UNTIL_REBOOT 标志合并使用。

- MOVEFILE_DELAY_UNTIL_REBOOT ——在 Windows NT 中执行时，函数并不马上进行移动操作，而是等下一次系统启动的时候再进行移动操作。
- MOVEFILE_REPLACE_EXISTING ——如果目标文件已经存在的话则将它替换掉，相比之下，MoveFile 函数无法替换已存在的目标文件。

MoveFileEx 函数还有个特殊用途：当标志指定 MOVEFILE_DELAY_UNTIL_REBOOT 的时候，lpNewFileName 参数可以指定为 NULL，在这种情况下，当下一次启动的时候，系统会删除 lpExistingFileName 指定的文件。MOVEFILE_DELAY_UNTIL_REBOOT 标志在 Windows 95 下不予支持。

3. 删除文件

删除文件使用 DeleteFile 函数，函数的语法很简单：

invoke	DeleteFile, lpFileName
--------	------------------------

lpFileName 指向一个包含要删除文件名的字符串。当函数在 Windows 95 下执行的时候，即使文件处在打开状态也可以成功删除，而在 Windows NT 下执行的时候，不能对一个已打开的文件进行删除，必须首先用 CloseHandle 函数关闭文件后才能执行成功。

10.3 驱动器和目录

先简单复习一下 Windows 中的逻辑驱动器、目录和路径等基本概念。

Windows 中的文件组织方式与 DOS 操作系统类似，也是采用分层次的结构：计算机中可以安装有多个物理驱动器，每个物理驱动器可以分为多个主分区和扩展分区，每个主分区就是一个逻辑驱动器，而每个扩展分区可以划分成多个逻辑驱动器，逻辑驱动器组成了我们熟悉的 C 盘与 D 盘等盘符。

如图 10.4 所示，假如计算机上安装了 2 个硬盘和 2 个光驱，每个硬盘都分为一个主分区和一个扩展分区，其中第一个硬盘的扩展分区中又分为 3 个逻辑驱动器，第二个硬盘的扩展分区分成 2 个逻辑驱动器，那么计算机中的逻辑驱动器就会从 C 盘一直排列到 K 盘为止。

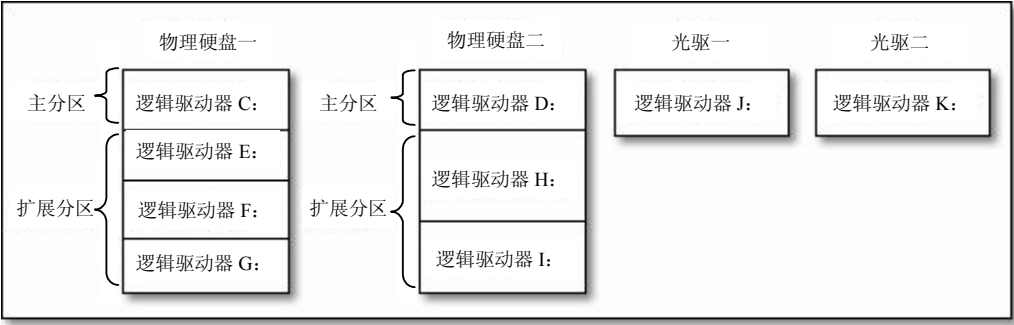


图 10.4 逻辑分区的分配

对于每个逻辑驱动器，可以给它取一个标号叫做“卷标”（Volume Label），卷标是当做一个目录项存放在逻辑驱动器的根目录中的。在每个逻辑驱动器中可以有多个文件，文件可以存放在各个目录中，目录是按照多层树状结构来安排的，每个逻辑驱动器中有个顶层目录叫做“根目录”，根目录下可以安排多个子目录，每个子目录中也可以包含多个下层子目录，一个逻辑驱动器中能够存放文件和子目录的数量只受限于驱动器的空间大小。

虽然同一个子目录中的文件名必须是惟一的，但不同的子目录中可以存在同名的文件，所以只有文件名的话并不能惟一确定一个文件，要惟一确定文件还需要指出文件的位置，除了指出文件位于的逻辑驱动器外，还要指出从根目录开始一直到文件所在目录为止的所有子目录名，这就是路径。

对一个进程来说，Windows 维护一个当前驱动器，并为每个逻辑驱动器维护一个当前路径，如果不指定路径，表示要操作的文件就位于当前驱动器的当前路径下。如果要操作非当前路径下的文件，就必须明确指出包含全路径的文件名。比如，指定一个文件 System.ini，如果当前目录下有这个文件，那么操作的对象就是这个文件，如果当前目录下并没有这个文件，即使其他目录中存在多个同名的文件，程序也无法知道它究竟对应哪个文件。

Win32 中有一部分函数专门用来完成与逻辑驱动器及目录有关的操作，在本节中将具体讨论这些函数的使用方法。

10.3.1 逻辑驱动器操作

1. 卷标操作

为一个驱动器创建、修改，以及删除卷标都使用 SetVolumeLabel 函数。如果要创建或修改卷标（如果原来没有卷标则为创建，原来存在卷标则为修改）可以这样使用：

szPath	db	'c:\',0
szVolume	db	'System',0
invoke	SetVolumeLabel,addr szPath,szVolume	

本例中，C 盘的卷标会被设置为 System，第一个参数指出了要设置卷标的逻辑驱动器的根目录，如果要设置 C 盘的卷标，目录名既不能写为“c:”也不能写为“c:\windows”，必须写为“c:\”，否则函数调用会失败；第二个参数则指向包含卷标字符串的缓冲区。

删除一个逻辑驱动器的卷标有两种方法。

方法一：

szPath	db	'c:\',0
szVolume	db	0
invoke	SetVolumeLabel, addr szPath, szVolume	

方法二：

szPath	db	'c:\',0
invoke	SetVolumeLabel, addr szPath, NULL	

这两种方法的执行结果是一样的。如果函数执行成功，返回值是 TRUE，否则返回 FALSE。
获取卷标可以使用下面介绍的 GetVolumeInformation 函数。

2. 逻辑驱动器的检测

要检测系统中当前存在多少个逻辑驱动器可以使用 GetLogicalDrives 函数，函数返回了所有可用的盘符。GetLogicalDrives 函数没有输入参数，它返回一个 32 位的整数，用其中的每一位代表是否存在一个逻辑驱动器。由于系统中可用的盘符仅有 26 个（A:~Z:），所以 32 位已经可以反映出所有的逻辑驱动器，以及它们的盘符分布情况了，返回值的第 0 位到第 25 位分别代表驱动器 A:~Z: 是否存在，如系统中存在 A, C, D, E 和 F 5 个逻辑驱动器的时候，返回值的二进制数值为 000000000000000000000000111101b，也就是十六进制的 0000003dh。

如果认为 GetLogicalDrives 函数返回的数据要进行位测试比较麻烦，可以使用另一个函数：GetLogicalDriveStrings，这个函数返回字符串类型的逻辑驱动器列表：

invoke	GetLogicalDriveStrings, dwBufferSize, lpBuffer	
--------	--	--

lpBuffer 指向一个缓冲区，函数在这里返回“A:\”，0，“B:\”，0，“C:\”，0，0 格式的字符串，凡是存在的逻辑驱动器都会列在这个字符串中，字符串列表以一个附加的 0 结束；dwBufferSize 指出缓存区的大小，如果缓冲区不够大，后面的数据会被截尾。

获取了逻辑驱动器的分布情况后，有时候还必须了解某个逻辑驱动器的类型，因为它可能是各种类型的盘——软盘、硬盘、光驱和内存中的虚拟盘等都是以逻辑驱动器的模样出现的。虽然文件操作函数中可以不必理会文件究竟位于什么样的驱动器上，只要指定全路径的文件名就可以透明地工作，但有时候必须检测盘的类型。比如，希望对软件进行保护，要求文件必须位于光盘上，那么就需要检查文件所在的逻辑驱动器是否是光盘；另外，需要建立一个临时文件的时候，如果建立在只读的光盘上是不会成功的，为了保证创建成功，需要预先检测一下程序是否运行于硬盘上。

检测驱动器类型的工作可以用 GetDriveType 函数来完成：

szPath	db	'c:\',0
invoke	GetDriveType, addr szPath	

该函数惟一的一个参数指向存放有逻辑驱动器根目录的字符串的缓冲区，函数的返回值是逻辑驱动器的类型，它可能是下面取值中的一种：

- 0——驱动器类型无法检测。
- 1——指定的根目录不存在。
- DRIVE_REMOVABLE——可移动介质，如软盘。
- DRIVE_FIXED——固定盘，如硬盘中的逻辑驱动器。
- DRIVE_REMOTE——远程驱动器，如网络上映射的驱动器。
- DRIVE_CDROM——光盘。
- DRIVE_RAMDISK——内存虚拟盘。

如果需要更详细的情况，可以使用 `GetVolumeInformation` 函数，这个函数可以返回逻辑驱动器的卷标、序列号和文件系统类型等属性：

```
invoke    GetVolumeInformation, lpRootPathName, \
          lpVolumeNameBuffer, dwVolumeNameSize, \
          lpVolumeSerialNumber, lpMaximumComponentLength, \
          lpFileSystemFlags, \
          lpFileSystemNameBuffer, dwFileSystemNameSize
```

参数 `lpRootPathName` 指向需要检测的驱动器根目录字符串，如果要检测的是网络上的驱动器，那么字符串可以是“\\服务器名\共享名”格式。

后面的各个参数指向一些用来返回数据的缓冲区。

`lpVolumeNameBuffer` 指向一个字符串缓冲区，用来返回驱动器的卷标，缓冲区的长度由 `dwVolumeNameSize` 参数指出。

`lpVolumeSerialNumber` 指向一个双字变量，函数在这里返回逻辑驱动器的序列号。序列号是驱动器被格式化的时候由系统随机生成的一个 32 位数，它保存在位于驱动器第一个扇区的引导记录中。在程序的运行中检测并记录软盘的序列号就可以检测到软盘是否被更换。

`lpMaximumComponentLength` 指向一个双字变量，函数在这里返回最大允许的文件名长度，在 Windows 系统中，一般这个数值是 255。

`lpFileSystemFlags` 也指向一个双字变量，函数在这里返回一些逻辑驱动器的属性标志，返回值可能是下面数值的组合：

- FS_CASE_IS_PRESERVED——文件系统在保存文件名的时候保持它的大小写（如 DOS 就不是这样，它把所有的文件名转换成大写后保存到目录区）。
- FS_CASE_SENSITIVE——支持区分大小写的文件名（在 Windows 中文件名不区分大小写，如 `Abc.exe` 和 `aBc.Exe` 指的是同一个文件）。
- FS_UNICODE_STORED_ON_DISK——允许存放 Unicode 格式的文件名。
- FS_PERSISTENT_ACLS——支持 ACL（访问控制列表），ACL 用于安全性管理，它是一个为个人或组委派或否认特定访问权限的条目列表。NTFS 文件系统支持 ACL，而 FAT 系统不支持。

- FS_FILE_COMPRESSION——支持文件压缩。
- FS_VOL_IS_COMPRESSED——支持卷压缩。

lpFileSystemNameBuffer 指向一个字符串缓冲区，用来接收文件系统字符串，函数在这里返回类似于“FAT”、“FAT32”或“NTFS”类型的字符串，dwFileSystemNameSize 参数指出了这个缓冲区的长度。

检测逻辑驱动器剩余空间的 GetDiskFreeSpace 函数也是一个常用的函数，它的用法如下：

```
invoke    GetDiskFreeSpace, lpRootPathName, \
          lpSectorsPerCluster, lpBytesPerSector, \
          lpNumberOfFreeClusters, lpTotalNumberOfClusters
```

同样，参数 lpRootPathName 指向需要检测的驱动器根目录字符串，后面的参数指向一些双字变量，用来接收返回的数据：

- lpSectorsPerCluster 参数——返回每簇的扇区数。
- lpBytesPerSector 参数——返回每扇区的字节数。
- lpNumberOfFreeClusters 参数——返回驱动器中未使用的簇的数量。
- lpTotalNumberOfClusters 参数——返回驱动器中簇的总数。

驱动器的总容量可以通过算式计算出来：簇总数×每簇扇区数×每扇区字节数，驱动器中空闲的字节数则等于：未使用的簇×每簇扇区数×每扇区字节数。

10.3.2 目录操作

1. 创建和删除目录

创建目录使用 CreateDirectory 函数，例如：

```
szDir     db    'c:\dir1\dir2',0
...
invoke    CreateDirectory, addr szDir, NULL
```

这两句代码在 c:\dir1 目录下创建一个名为 dir2 的新子目录。如果创建成功，函数返回 TRUE，否则返回 FALSE。

在创建目录的时候要注意几个要点：首先是要创建目录的上层目录必须存在，上面的例子中，假如 c:\dir1 目录不存在，函数不会创建 dir2 目录；其次是与新建目录同名的目录或文件不能存在，假如 c:\dir1 目录中已经存在一个名为 dir2 的目录或文件，那么创建工作就会失败。由于文件名和目录名在磁盘目录区中的存放格式是一样的，惟一的不同的是目录项的属性不同（在 10.2.3 节中，已经发现判别找到的目录项是文件还是目录的惟一办法就是检测 FILE_ATTRIBUTE_DIRECTORY 属性），所以，连同名文件的存在也是不允许的。

删除目录使用 RemoveDirectory 函数，删除上面创建的 dir2 目录的方法是：

```
szDir     db    'c:\dir1\dir2',0
...
invoke    RemoveDirectory, addr szDir
```

如果删除成功，函数返回 TRUE，否则返回 FALSE。

删除目录也要注意几个要点：首先，被删除的是参数中指出的最后一级目录，如前面代码只删除 dir2 目录，而不会将 dir1 目录和 dir2 目录一起删掉；其次，在删除目录之前必须删除目录中的所有文件，以及子目录，函数无法删除一个不为空的目录；最后，函数执行以后目录是被“真正”删除掉了，不会像用手工删除一样还可以在回收站中恢复回来。

2. 一些特殊目录

这里列出了 Windows 操作系统中的一些特殊目录：

- 当前目录——所有未指定路径的文件名均默认使用这个目录。
- Windows 目录——Windows 操作系统的安装目录。
- 系统目录——Windows 安装目录下存放系统文件的目录，Windows 9x 下是 System 目录，Windows NT 下是 System32 目录。
- 临时目录——存放临时文件的目录，系统可以在磁盘空间不足的时候自动删除里面的文件。

这些目录在编程的时候是经常需要检测的，如要编写系统程序的时候常常要把 dll 文件拷贝到系统目录或 Windows 目录中去；而要建立临时文件的时候最好使用系统指定的临时目录，以便自动回收使用的空间。Win32 中专门设置了几个函数来获取这些目录的位置：

invoke	GetCurrentDirectory, dwBufferSize, lpBuffer	;获取当前目录
invoke	GetTempPath, dwBufferSize, lpBuffer	;获取临时目录
invoke	GetWindowsDirectory, lpBuffer, dwBufferSize	;获取 Windows 目录
invoke	GetSystemDirectory, lpBuffer, dwBufferSize	;获取系统目录

参数 lpBuffer 指向一个缓冲区，用来接收返回的路径字符串，dwBufferSize 指出了缓冲区的大小，一般把缓冲区的大小设置为 MAX_PATH。

这几个函数有些奇怪是：GetWindowsDirectory 和 GetSystemDirectory 函数的参数和通常的习惯一致，把缓冲区指针 lpBuffer 放在前面，而 GetCurrentDirectory 和 GetTempPath 函数却把 dwBufferSize 参数放在前面，不注意的话很容易搞错；另外，GetTempPath 函数返回的路径的最后竟然包括“\”，在笔者的 Windows XP 操作系统中，它的返回值是“C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\”，而其他 3 个函数返回的路径最后并不带“\”，读者在路径后面添加文件名的时候最好首先检测一下字符串的最后是不是已经包含了一个“\”字符，否则会构造出一个类似于“c:\temp\abc.dat”之类的无效的文件名。

当程序运行的时候，默认当前路径是执行程序所在的目录，但这不是绝对的，比如使用 GetOpenFileName 等函数弹出一个打开文件的系统对话框，用户在里面指定了其他目录后，当前路径就会被设置到那个目录中。

读者也可以自己调用 SetCurrentDirectory 函数修改当前路径：

szDir	db	'c:\dir1\dir2', 0
...		
invoke	SetCurrentDirectory, addr	szDir

这几句代码将当前路径设置到“c:\dir1\dir2”目录中。

10.4 内存映射文件

首先回过头来看看 10.2.1 节的 FormatText 例子中读文件的操作，与在 DOS 操作系统下所有文件操作代码的结构一样，例子中用分块读入的办法读取文件，每次读取的内容受限于缓存区的大小，对上一次读入的内容处理完毕后，程序才能继续读入下一块内容，代码结构如下：

```
.while    TRUE
    调用 ReadFile 读取文件
    .break    .if 读取的字节数为 0 (到达文件尾部)
    从缓冲区中取出数据并处理
.endw
```

使用这种结构处理文件的严重缺陷就是缓冲区边界的处理问题，读者可以尝试将 FormatText 程序的功能改造成清除每一行尾部的多余空格，例如：假如某一行的内容是“abc def ”则转换成“abc def”，这样，当缓冲区结束的地方刚好是连续的空格的时候，就必须先保存这部分内容，等继续将后续数据读到缓冲区后才能判别这是行尾的空格还是两个单词之间的空格；把这个问题再扩展开来，假如某一次遇到的连续空格特别多，长度等于 3 个、4 个甚至很多个缓冲区的长度，又要如何处理呢？这就已经涉及有名的边界判断问题了，实际上程序中许许多多的错误就是由此引起的。

解决这个问题最简单的办法就是一次性把文件全部读进内存，这样就不存在边界问题了，但在 DOS 操作系统下，程序能用的最大内存一般只有几百 KB，又有多少个程序能保证自己要处理的文件一定能够一次性全部读进内存呢？毕竟包括操作系统在内的所有可寻址的地址空间只有 1 MB。

在 Windows 中，每个进程可以自由使用的地址空间达到了 2 GB，这就为将整个文件读进内存打好了基础，但程序还是需要预先分配一块大小等于文件长度的内存块，所以限制还是不少，因为在内存分配一节中已经发现，当要分配的内存块大小远远超过可用物理内存的时候，分配工作并不一定会成功。

Win32 中内存映射文件的引入，使这一类问题得到较好的解决，更使 Win32 程序员们信心大增，笔者也认为，内存映射文件是 Win32 中最有实用价值的新特征之一。

10.4.1 内存映射文件简介

1. 内存映射文件的概念

内存映射文件提供了一组独立的函数，使应用程序能够通过内存指针像访问内存一样对磁盘上的文件进行访问。通过内存映射文件函数可以将磁盘上文件的全部或部分映射到进程虚拟地址空间的某个位置，一旦完成了映射，对文件内容的访问就如同在该地址区域内直接对内存访问一样简单。这样，向文件中写入数据的操作就是直接对内存进行赋值，而从文件的某个特定位置读取数据也就是直接从内存中取数据。

当内存映射文件提供了对文件某个特定位置的直接读写时，真正对磁盘文件的读写操作是由系统底层处理的。而且在写操作时，数据也并非在每次操作时都即时写入到磁盘，而是通过缓冲处理来提高系统的整体性能。

使用内存映射文件的好处之一是系统对所有的数据传输都是通过 4 KB 大小的数据页面来实现的，这意味着一些小的文件操作将被缓冲入一次大的操作之中，也就是说首次存取文件中某段数据的时候，会引发一次磁盘操作并将数据所在的一个页面全部读入，到以后对附近的数据进行操作时，所需的数据已经被前一次的页面操作读入到内存，无需再进行一次磁盘操作，从而提高了系统的性能。

另一个好处是程序代码以标准的内存地址形式来访问文件数据，按页面大小周期性从磁盘读入数据的操作发生在后台，由操作系统底层来实现，这个过程对应用程序是完全透明的。虽然用内存映射文件最终还是要将文件从磁盘读入内存，实质上并没有省略掉什么操作，整体性能可能并没有获得什么提高，但是程序的结构将会从中受益，缓冲区边界等问题将不复存在。而且，对文件内容更新后的写入操作也由操作系统自动完成，由操作系统来判断内存中的页面是否为脏页面并仅将脏页面写入磁盘，比程序自己将全部数据写入文件的效率要高了很多。

2. 内存映射文件的实现原理

Windows 使用的是页式虚拟存储管理，在 Windows 中，地址空间中的每个页面在任一给定时刻都可以是三种状态之一：空闲的、保留的或者是已经提交物理内存的。这些页面根据需求由操作系统交换进内存或换出内存。当内存中的某个页面不再需要时，操作系统将取消原来拥用该页面的应用程序对它的控制权，并释放该页面以供其他应用程序使用；当该页面再次成为需求页面时，它将被从物理存储器中重新读入内存，物理存储器既可以是物理内存，也可以是磁盘上的页文件。

内存映射文件的实现基于同样的原理，内存映射文件是 Windows 内部已有的内存管理组件的一个扩充，与实现虚拟内存一样，内存映射文件保留了一个地址空间的区域，并根据需要将物理存储器提交给该区域。它们之间的区别在于，当内存映射文件用来存取一个磁盘文件的时候，它提交的物理存储器就来自于这个文件。

不仅应用程序使用内存映射文件来访问磁盘上的数据文件，Windows 操作系统同样使用内存映射文件加载和执行 exe 和 dll 文件，这样可以大大节省页文件空间和应用程序启动运行所需的时间。如图 10.5 所示，对于每个进程，系统将可执行的代码页提交到磁盘中的可执行文件中，而数据页

（包括进程的静态数据段以及动态分配的内存）则被提交到虚拟内存中。

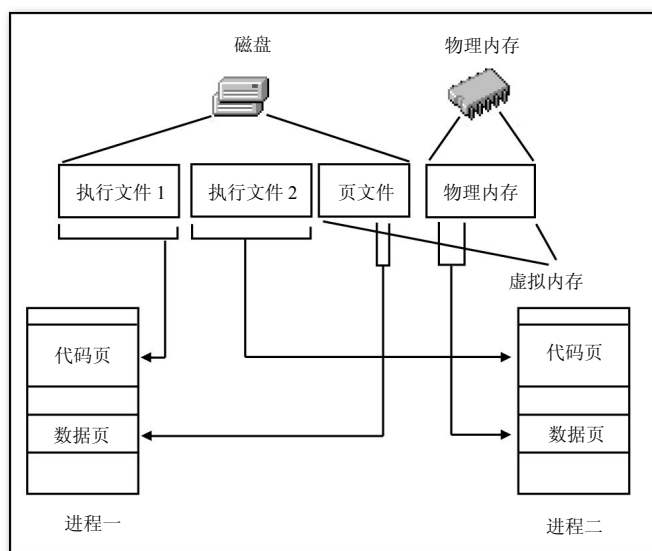


图 10.5 用内存映射文件加载执行文件

除了加载文件，使用内存映射文件也可以在同一台计算机上运行的多个进程之间共享数据，而且内存映射文件是多个进程互相进行通信的最有效的方法。那么如何实现数据共享呢，其实原理很简单，如图 10.6 所示，对于不同进程间共享的数据页，只要将它们提交到虚拟内存的同样页面就可以了，这样，当一个进程改变了数据页的内容时，通过分页映射机制，其他进程的共享数据区的内容就会同时改变，因为它们实际上存储在同一个地方。

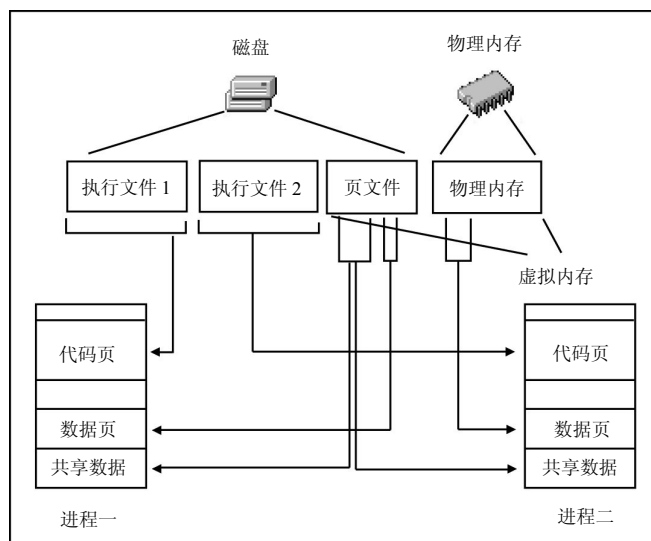


图 10.6 用内存映射文件实现进程间共享数据

10.4.2 使用内存映射文件

1. 内存映射文件函数

内存映射文件函数包括：CreateFileMapping，OpenFileMapping，MapViewOfFile，UnmapViewOfFile 和 FlushViewOfFile。

使用内存映射文件的步骤分为两步，第一步是使用 CreateFileMapping 创建一个内存映射文件对象。这个步骤决定了使用内存映射文件的用途——究竟是在磁盘文件上建立内存映射文件还是在页文件中建立进程间共享的映射。CreateFileMapping 函数的用法是：

```
invoke    CreateFileMapping, hFile, lpFileMappingAttributes, \
          flProtect, dwMaximumSizeHigh, dwMaximumSizeLow, lpName
. if      eax
          mov          hFileMap, eax
. endif
```

函数的第一个参数 hFile 指定一个文件句柄。如果句柄是属于一个已经打开的文件的，那么内存映射文件将在这个文件上面建立；如果需要建立存在于页文件中的内存映射文件供不同进程共享，那么 hFile 指定为 -1。

lpFileMappingAttributes 参数指向一个 SECURITY_ATTRIBUTES 结构，用来定义内存映射文件对象是否是可继承的。这个结构在文件打开函数中也曾经用到过，如果句柄不需要继承，可以把这个参数设置为 NULL。

第三个参数 flProtect 指定该内存映射文件的保护类型，它可以是以下取值：

- PAGE_READONLY——内存映射文件提交的内存页面是只读的，为了使用此标志获得对应的读权限，在用 CreateFile 函数打开文件获得 hFile 句柄时必须相应指定 GENERIC_READ 标志。
- PAGE_READWRITE——内存映射文件提交的内存是可读写的。为了使用此标志，在用 CreateFile 函数打开文件获得 hFile 句柄时，必须同时指定 GENERIC_READ 标志和 GENERIC_WRITE 标志。
- PAGE_WRITECOPY——内存映射文件提交的内存可以有 Copy on Write 属性。为了使用此标志，在用 CreateFile 函数打开文件获得 hFile 句柄时，必须同时指定 GENERIC_READ 标志和 GENERIC_WRITE 标志。

dwMaximumSizeHigh 和 dwMaximumSizeLow 参数则组合指定了一个 64 位的内存映射文件的长度。当内存映射文件用于磁盘文件的时候，如果这个长度大于磁盘文件的长度，那么磁盘文件将被扩展到这个长度；如果小于磁盘文件长度，那么只能存取磁盘文件的一部分。一种简单的方法是将这两个参数全部设置为 0，那么内存映射文件的大小将被自动调整到磁盘文件的大小。

最后一个参数 lpName 指定一个字符串，用来给定内存映射文件的名称。当内存映射文件用于磁盘文件的时候，不需要给它起名；如果用于在进程间共享内存，那么必须为该对象命名，因为在其他进程中只有使用这个名称才能打开这个内存映射文件对象，该名字字符串不能和其

他进程已创建的对象同名。

当一个进程创建内存共享文件用于和其他进程共享的时候，其他进程不能再使用 `CreateFileMapping` 函数去创建同样的内存映射文件对象，而是要用 `OpenFileMapping` 函数去打开已创建好的对象。`OpenFileMapping` 函数的用法是：

```

invoke    OpenFileMapping, dwDesiredAccess, bInheritHandle, lpName
.if       eax
    mov     hFileMap, eax
.endif

```

这里的 `lpName` 参数指向的名字就是创建对象时使用的名字，`dwDesiredAccess` 参数指定保护类型，它可以是以下的取值：

- `FILE_MAP_WRITE`（或 `FILE_MAP_ALL_ACCESS`）——可写属性。
- `FILE_MAP_READ`——可读属性。
- `FILE_MAP_COPY`——Copy on write 属性。

注意：`FILE_MAP_ALL_ACCESS` 等于 `FILE_MAP_WRITE` 属性，并不同时包括 `FILE_MAP_READ` 属性。如果 `CreateFileMapping` 函数或 `OpenFileMapping` 函数执行成功，返回的是内存映射文件句柄，这个句柄可以用在后面的函数中，如果执行失败则返回 `NULL`。

使用内存映射文件的第二个步骤是创建内存映射文件的一个视图。获得内存映射文件对象的句柄后，就可以使用它在进程的地址空间中映射该文件的一个视图，该操作可以视为给需要映射的文件内容分配线性地址空间，并将线性地址和文件内容对应起来，这样程序就可以通过存取线性地址来存取文件。

视图可以任意映射或取消映射。当一个文件的视图被映射时，系统仅为它分配足以覆盖文件视图的连续地址空间，并不马上将它提交到当做物理存储器的文件中去，当第一次读写内存页面中任一地址的时候，系统才真正分配一个对应于视图页面的物理内存页面，所以映射视图的速度是相当快的。

`MapViewOfFile` 函数用来映射内存映射文件的一个视图。这个函数的用法是：

```

invoke    MapViewOfFile, hFileMap, dwDesiredAccess, \
            dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap
.if       eax
    mov     lpMemory, eax
.endif

```

参数 `hFileMap` 就是前两个函数返回的内存映射文件对象的句柄，`dwDesiredAccess` 参数指定保护类型，可能的取值同样是 `FILE_MAP_WRITE`，`FILE_MAP_READ` 或 `FILE_MAP_COPY`。

一个视图可以映射到整个文件，也可以映射到磁盘文件的一部分。需要映射的起始位置可以由 `dwFileOffsetHigh` 和 `dwFileOffsetLow` 指定，这两个参数组合成一个 64 位的偏移量，用来指定视图的基地址是从文件的哪个位置开始映射。`dwNumberOfBytesToMap` 参数指定要映射的字节数，如果 `dwNumberOfBytesToMap` 参数设置为 0，那么映射的是整个文件，同时偏移地址被忽略。如果映射成功，函数返回一个地址，存取这个地址指定的内存块就相当于存取文件的内

容了。如果映射失败，则函数返回 NULL。

当不再使用内存映射文件后，可以通过 `UnmapViewOfFile` 函数撤销映射并使用 `CloseHandle` 函数关闭内存映射文件对象句柄：

```
invoke    UnmapViewOfFile, lpMemory
invoke    CloseHandle, hFileMap
```

当对视图中的内存进行修改后，系统会在视图撤销映射或文件映射对象被删除时自动将数据写到磁盘上，但程序也可以根据需要对文件的修改立即写到磁盘上，该功能是由函数 `FlushViewOfFile` 提供的：

```
invoke    FlushViewOfFile, lpMemory, dwFileSize
```

该函数将从指定地址开始、指定大小的数据块中的脏页面写到磁盘，指定的内存范围必须位于视图的边界之内。

2. 使用内存映射文件读写文件

通过上一节的讨论，读者已经知道使用内存映射文件读写文件的步骤为：

(1) 调用 `CreateFile` 打开想要映射的文件，得到 `hFile`。

(2) 调用 `CreateFileMapping` 函数生成一个建立在 `CreateFile` 函数创建的文件对象基础上的内存映射对象，得到 `hFileMap`。

(3) 调用 `MapViewOfFile` 函数把整个文件的一个区域或者整个文件映射到内存中。得到指向映射到内存的第一个字节的指针 `lpMemory`。

(4) 用该指针来读写文件。

(5) 调用 `UnmapViewOfFile` 来解除文件映射，传入参数为 `lpMemory`。

(6) 调用 `CloseHandle` 来关闭内存映射文件，传入参数为 `hFileMap`。

(7) 调用 `CloseHandle` 来关闭文件，传入参数为 `hFile`。

现在，将 `FormatText` 例子中的读文件部分改成使用内存映射文件的方法，将其中的 `_ProcFile` 子程序改成下面的样子后，使用效果是一样的：

```
_ProcFile    proc
               local    @hFile, @hFileNew, @hFileMap, @lpMemory, @dwFileSize
               local    @szNewFile[MAX_PATH]:byte
               local    @szBuffer[512]:byte

;*****
; 打开文件
;*****
               invoke    CreateFile, addr szFileName, GENERIC_READ, FILE_SHARE_READ, \
                           0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
               .if      eax == INVALID_HANDLE_VALUE
                   ; 显示出错提示
                   ret
               .endif
```

```

        mov     @hFile, eax
        invoke  GetFileSize, @hFile, NULL
        mov     @dwFileSize, eax
;*****
; 建立内存映射文件
;*****
        invoke  CreateFileMapping, @hFile, NULL, PAGE_READONLY, 0, 0, NULL
        .if     ! eax
            ; 显示出错提示
            jmp     _Ret1
        .endif
        mov     @hFileMap, eax
        invoke  MapViewOfFile, eax, FILE_MAP_READ, 0, 0, 0
        .if     ! eax
            ; 显示出错提示
            jmp     _Ret2
        .endif
        mov     @lpMemory, eax
;*****
; 创建输出文件
;*****
        invoke  lstrcpy, addr @szNewFile, addr szFileName
        invoke  lstrcat, addr @szNewFile, addr szNewFile
        invoke  CreateFile, addr @szNewFile, GENERIC_WRITE, \
            FILE_SHARE_READ, \
            0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
        .if     eax == INVALID_HANDLE_VALUE
            ; 显示出错提示
            jmp     _Ret3
        .endif
        mov     @hFileNew, eax
;*****
; 在映射的内存中处理文件
;*****
        invoke  _FormatText, @lpMemory, @dwFileSize, @hFileNew
        invoke  CloseHandle, @hFileNew
_Ret3:
        invoke  UnmapViewOfFile, @lpMemory
_Ret2:
        invoke  CloseHandle, @hFileMap
_Ret1:
        invoke  CloseHandle, @hFile
        ret
_ProcFile endp

```

读者可以在所附光盘的 Chapter10\FormatText\FileMap 目录中找到修改后的源程序。对比原来的程序可以看出，程序中分步将数据读取到缓冲区的循环不见了，取而代之的是对一个长度等于文件长度的大缓冲区进行的单次操作，这样，缓冲区边界判断之类的难题就无形中消失了。

3. 使用内存映射文件在进程间共享数据

先来看一个例子，例子在所附光盘的 Chapter10\MMFShare 目录下。首先多次执行目录中的 MMFShare.exe 文件，然后尝试在不同执行副本中的编辑框中输入字符，读者马上可以发现，不管在哪个副本中输入字符，所有副本的文本框中都会被设置成刚刚输入的内容（如图 10.7 所示），这就是用内存映射文件实现的。



图 10.7 使用 MMF 进行进程间共享

使用内存映射文件在进程间共享数据的步骤如下：

(1) 调用 `OpenFileMapping` 打开一个命名的内存映射文件对象，得到 `hFileMap`。如果打开成功则跳到步骤 (3)，如果打开不成功，则表示本进程是执行的第一个副本，那么继续执行步骤 (2)。

(2) 调用 `CreateFileMapping` 函数创建一个命名的内存映射对象, 得到 `hFileMap`。

(3) 调用 MapViewOfFile 函数映射对象的一个视图, 得到指向映射到内存的第一个字节的指针 lpMemory。

(4) 用该指针来读写共享的内存区域。

(5) 调用 `UnmapViewOfFile` 来解除视图映射，传入参数为 `lpMemory`。

(6) 调用 CloseHandle 来关闭内存映射文件，传入参数为 hFileMap。

上面的步骤与映射普通的磁盘文件相比，少了打开和关闭文件的步骤，但多了一个 `OpenFileMapping` 的步骤。还有一个区别在于，建立内存映射文件对象的时候使用的不是文件句柄，而是使用命名的方法。

具体的实现方法参见例子文件 MMFS_{Share}.asm 中的源代码:

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include      windows.inc
include      user32.inc
includelib   user32.lib
include      kernel32.inc
includelib   kernel32.lib

```

哭

[illegible]

对应的资源脚本文件 MMFSshare.rc 如下:

```
#include <resource.h>
#define ICO_MAIN 1000
#define DLG_MAIN 100
#define IDC_TXT 101
#define IDC_INFO 102
ICO_MAIN ICON "Main.ico"
DLG_MAIN DIALOG 229, 208, 211, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "内存映射文件共享"
FONT 9, "宋体"
{
    LTEXT "请执行本程序的多个拷贝，并尝试在下面输入文本：", -1, 7, 8, 196, 8
    EDITTEXT IDC_TXT, 7, 22, 197, 12, ES_AUTOHSCROLL | WS_BORDER | WS_TABSTOP
    LTEXT "", IDC_INFO, 8, 41, 196, 8
}
```

程序的结构异常简单，首先在用做主界面的对话框的初始化消息中创建内存映射文件（在 `_CreateMMF` 子程序中完成），并建立一个定时器，用来不停地将共享区域的内容设置到文本框中。在这里，内存映射文件对象的名称是 `"MMF_Share_Example"`，别的进程只要知道这个名称，就可以使用它。

程序在 `WM_COMMAND` 消息中检测编辑框的输入动作，如果用户在编辑框中输入了字符，那么马上将编辑框的内容取到共享区域中，这样，其他进程在定时器消息中就可以马上将这个内容在自己窗口的文本框中反映出来。

在退出的时候，程序在 `_CloseMMF` 子程序中调用 `UnmapViewOfFile` 和 `CloseHandle` 函数来关闭内存映射文件，并在撤销定时器后退出。

当程序运行多个副本的时候，内存映射文件是由首先运行的副本建立的，但是在退出的时候，即使首先运行的副本先退出（也就是说创建内存映射文件的副本首先退出），其他副本之间的通信也不受影响，这就是说，这时内存映射文件还是存在的。实际上，系统为进程间共享的内存映射文件对象维护一个计数器，每次有进程打开内存映射文件对象的时候，计数器加 1，关闭的时候减 1，只有当计数器减到零的时候，内存映射文件才真正被释放，所以程序中关闭内存映射文件的时候并不需要考虑别的程序是否还在使用它。

第 11 章

动态链接库和钩子

动态链接库在 Windows 系统中无处不在，在前面的章节中介绍过很多的 API 函数，这些函数全部都是以动态链接库的方式提供的，可以说，在不知不觉中，已经在每个例子中都用到了动态链接库。实际上，在 Win32 编程中不使用动态链接库是几乎不可能的，因为 Windows 提供给程序员的几乎所有功能都驻留在动态链接库里面。

在本章中，将介绍如何编写动态链接库，并更全面地探讨动态链接库的使用方法，包括以不同的方法装入动态链接库和以不同的方法调用其中的函数等。

除了可以使用动态链接库来发布产品外，Windows 中还有一些功能是必须使用动态链接库的，如使用钩子函数等。由于 Windows 钩子的使用和动态链接库密切相关，所以将这部分的内容也放在本章中介绍。

11.1 动态链接库

11.1.1 动态链接库的概念

在 DOS 环境下编过程序的读者一定知道静态库的含义——程序员将实现各种功能的代码写成一个个子程序（函数），编译成 obj 文件后，用 Lib.exe 工具将多个 obj 文件组合成一个 lib 文件，当程序中要用到这些函数的时候，只需要指定函数名称，链接器就可以从库中抽出对应的子程序代码插入到可执行文件中去，这样就可以不必一遍遍地重写相同的功能代码，这种链接方法就是静态链接。

静态链接的缺点显而易见，如果有多个程序用到库中的同样函数，那么所有这些可执行文件中都会包含一份同样的代码，对于几乎每个程序都必须使用的常用函数来说，如果硬盘上有一万个程序用到这个函数，那么就存在一万份相同的代码，这显然是很浪费空间的。静态链接的另外一个缺点是：如果某个函数因为发现有错或更新算法等种种原因需要升级时，必须把所有用到此函数的可执行文件都找回来重新编译一遍，遗漏的程序中存在的还是旧版本的代码。

另外，从内存使用的方面考虑，DOS 操作系统是单任务的操作系统，每时每刻只能有一个程序在运行，所以使用静态链接浪费的空间仅表现在磁盘空间的浪费上；而 Windows 操作系统

是多任务的，内存中会同时装入多个程序的代码，如果使用静态链接的话，意味着有多份相同的代码被装入内存，这种浪费的代价将是更昂贵的。

Windows 的解决办法就是使用动态链接库，动态链接库从表面上看也是提供了一大堆通用的函数，也可以被多个程序使用，但它和静态库在使用上有很多的不同点。

静态库仅在链接的时候使用，链接完成后，可执行文件就可以脱离库文件单独使用了。而动态链接库中的代码在程序链接的时候并不会被插入到可执行文件中，在程序运行的时候才将整个库的代码调入内存，所以称为“动态链接”。如果有多个程序用到同一个动态链接库，Windows 在物理内存中只保留一份库的代码，仅通过分页机制将这份代码映射到不同进程的地址空间中，这样不管有多少程序在使用一个库，库代码实际占用的物理内存永远只有一份。当然，这时候库使用的数据段还是会被映射到不同的物理内存中，多少个程序在使用动态链接库就会有多少份数据段。动态链接库的工作方式在图 1.6 中就已经有所演示了。

如果应用程序要使用动态链接库中的函数，那么程序中必须包括库的名称和函数的名称，这是动态寻找对应函数所必需的，这些定位信息在编译和链接的时候被插入到可执行文件中。定位信息取自导入库文件，在前面这么多章的编程中我们已经多次用到了导入库文件。

动态链接库的缩写为 DLL，大部分动态链接库是以扩展名为 dll 的文件形式存在的，但并不是只有 dll 扩展名的文件才是动态链接库，系统中的某些 exe 文件、字体文件 (*.fon)、一些驱动程序 (*.drv 和 *.sys)、各种控件 (*.ocx) 和输入法模块 (*.ime) 等都是动态链接库。实际上，系统中大部分包含公用代码的模块——不管扩展名是什么——都有可能是动态链接库。

一个文件是否是动态链接库取决于它的文件结构，动态链接库文件和可执行文件同样使用标准的 PE 文件格式，仅文件头中的属性位不同而已，所以 exe 文件的一些特征也存在于动态链接库中，比如，在动态链接库中也可以定义并使用各种资源，可以导入并使用其他动态链接库中的函数等。

有一个最重要的概念一定要牢记：动态链接库是被映射到其他应用程序的地址空间中执行的，它和应用程序可以看成是“一体”的，动态链接库可以使用应用程序的资源，它所拥有的资源也可以被应用程序使用，它的任何操作都是代表应用程序进行的，当动态链接库进行打开文件、分配内存和创建窗口等操作后，这些文件、内存和窗口都是为应用程序所拥有的。

11.1.2 编写动态链接库

与前面一些例子程序相比，写动态链接库程序应该算是很简单的，程序中并不需要用到新的函数，只是在程序的结构上和链接时的选项有些区别而已。让我们通过一个简单的例子来说明，例子代码在所附光盘的 Chapter11\Dll\Dll 目录中，包括汇编源文件 Sample.asm 和定义文件 Sample.def。上一层子目录 Chapter11\Dll 中的另两个子目录存放的是使用汇编和 VC++ 调用这个 DLL 的程序，其中的内容将在下一节中分析。

Sample.asm 的内容如下：

```
.386
.model flat, stdcall
option casemap :none
```

386

一眼看上去，程序比较莫名其妙——入口的代码什么都没有做，仅返回一个 TRUE；也没有地方用到_IncCounter、_DecCounter 和_Mod 函数，这是为什么呢？请记住，d11 文件被设计为不是供自己使用的，而是被映射到其他应用程序的地址空间中代表“宿主”程序执行的，这些函数就是供其他程序使用的函数，对于“宿主”程序来说，虽然这些函数仅包含几行代码，但它们的“级别”和 User32.dll 中的 CreateWindowEx 与 DefWindowProc 等极其复杂的函数没有任何区别。

与可执行文件一样，动态链接库需要一个入口点，动态链接库的入口点是一个函数，函数的名称并不重要，例子代码中的入口函数命名为“DllEntry”，读者也可以把它取名为其他任何合法的名字，但入口函数的格式是有规定的。

```
                mov     eax, TRUE
            .else
                mov     eax, FALSE
            .endif
        .elseif  eax ==  DLL_THREAD_ATTACH
            ;为新的线程分配资源
        .elseif  eax ==  DLL_THREAD_DETACH
            ;为线程释放资源
        .elseif  eax ==  DLL_PROCESS_DETACH
            ;释放库使用的资源
        .endif
        ret
```

DllEntry

Endp

Windows 会传给入口函数 3 个参数，dwReason 参数的值表示本次调用的原因，它可能是下面的四种情况之一。

当 dwReason 的值是 DLL_PROCESS_ATTACH 的时候，表示动态链接库刚被映射到某个进程的地址空间，程序可以在这里进行一些初始化的工作，并返回 TRUE 表示初始化成功，返回 FALSE 表示初始化出错，这样库的装入就会失败。这给了动态链接库一个机会来阻止自己被装入。比如，库程序可以在这里申请并保留一些内存，如果申请失败的话就可以返回 FALSE 告诉 Windows，库无法正常工作。

当 dwReason 的值是 DLL_PROCESS_DETACH 的时候则相反，表示动态链接库将被卸载，库程序可以在这里进行一些资源的释放工作，如将初始化时申请的内存释放，将打开的文件关闭等。以 DLL_PROCESS_ATTACH 和 DLL_PROCESS_DETACH 值进行的调用在库的生命周期中只可能出现一次。

当 dwReason 的值是 DLL_THREAD_ATTACH 的时候，表示应用程序创建了一个新的线程。当某个线程正常终止的时候，dwReason 的值是 DLL_PROCESS_DETACH。如果应用程序不是以多线程方式工作的话，就不会有这两种原因的调用；反之，如果应用程序频繁地创建和结束线程，那么入口函数将不断被调用。

hInstDll 是动态链接库的模块实例句柄。当使用这个句柄来装入资源的时候，表示资源是定义在库文件中的。对于动态链接库来说，获取这个句柄的惟一途径就是在入口函数被调用的时候保存这个参数，如果在 DLL_PROCESS_ATTACH 时不将这个句柄保存下来的话，运行时可能就没有其他方法可以获取了。

dwReserved 参数是系统保留的参数，可以不必理会。



读者可能会问：不是可以通过 invoke GetModuleHandle, NULL 来获取模块实例句柄吗？是的，但是由于动态链接库是代表应用程序运行的，所以，如果在库中调用这个函数，得到的仍然是“宿主”程序的实例句柄，而不是库程序的实例句柄。

在例子程序中，不需要初始化工作，所以仅返回一个 TRUE，表示任何情况下，Windows 都

可以装入这个库文件。动态链接库有一种很“极端”的应用：纯资源库，这些库仅包含资源而没有任何的功能函数，如字体文件等，对于这些库来说，库中的全部代码仅是入口函数中用来返回 TRUE 的那几句，这是库能被正常装入所必需的代码。

2. 导出函数

与写普通的可执行文件相比，动态链接库的设计流程中多了一个文件，那就是定义文件 *.def，源代码目录中就有一个 Sample.def 文件，它的内容是：

```
EXPORTS
    _IncCounter
    _DecCounter
    _Mod
```

文件内容总共只有三行：一个 EXPORTS 关键字加上三个库中函数的名称，这是用来告诉链接器这三个函数需要导出，也就是说这三个函数可以被其他程序调用。动态链接库的文件格式是 PE 格式，每个 PE 格式文件的文件头中都可以有一个导出表，只有导出表中列出的函数才可以被其他程序调用，链接器根据 def 文件的内容在导出表中加入由 EXPORTS 关键字指定的函数名（导出表的详细说明请参考第 17 章）。

如果库文件中的函数没有在 def 文件中指定（如例子中的 _CheckCounter 函数），那么这个函数就仅能被库文件本身之中的代码调用，而无法在其他应用程序中使用，这是因为库文件的导出表中没有列出它的名称，这样其他程序根本不会知道它的存在。对于这些函数，可以把它们叫做私有函数。

3. 链接选项

为了生成动态链接库文件，在链接的时候必须使用合适的选项，来看看 Sample 库文件例子使用的 Makefile 文件：

```
DLL = Sample
ML_FLAG = /c /coff
LINK_FLAG = /subsystem:windows /D11

$(DLL).dll: $(DLL).obj $(DLL).def
    Link $(LINK_FLAG) /Def:$(DLL).def $(DLL).obj
.asm.obj:
    ml $(ML_FLAG) $<
.rc.res:
    rc $<
clean:
    del *.obj
    del *.exp
```

编译的时候，使用 Ml.exe 编译器的方法并没有什么不同，但是使用 Link.exe 链接程序的时候，必须使用 /D11 和 /Def 选项，/D11 选项告诉链接器输出文件的格式是动态链接库，/Def:filename.def 选项用来指定定义了导出函数名称的 def 文件名，在这个例子中，库文件中没有包含资源，如果包含资源的话，链接时还可以指定资源文件名，一个完整的链接参数如下所示：

Link /DLL /subsystem:windows /Def:filename.def filename.obj filename.res

4. 发布动态链接库

当使用 Link.exe 链接器完成链接工作后，链接器生成 3 个文件，它们分别以 dll, lib 和 exp 为扩展名，dll 文件就是动态链接库，而 lib 文件是供程序开发用的导入库，exp 文件是输出库文件，这是链接时的一个副产品，一般没有什么用途，我们可以直接将它删掉。

回想一下：当在汇编源程序中用到某个动态链接库中的函数时，在源文件的一开始就要用 `includelib` 语句指定动态链接库的导入库，这样链接的时候链接器才知道到哪个库中寻找指定的函数，如果开发的时候没有动态链接库的导入库文件，使用起来就比较麻烦了。

为了在开发其他程序的时候使用自己编写的动态链接库，就必须提供这个动态链接库的导入库文件，Link.exe 考虑了这一点，所以在生成 dll 文件的同时也生成了导入库文件。如果 dll 文件是作为最终应用程序的一部分发布的，可以仅发布 dll 文件；如果是当做组件供其他人做二次开发用的，那么开发者就应该为其他程序员同时提供 dll 文件和 lib 文件，并且根据情况提供不同语言使用的头文件，头文件中最好为每个导出函数写一个说明，包括函数的功能、参数的数量、类型和定义等，同时写上版权、版本号等信息，以便其他程序员参考使用。

例如，目录中还有一个 `Sample.inc` 文件，这就是为 `Sample.dll` 文件书写的供其他汇编程序员使用的头文件，它的内容如下：

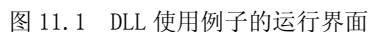
```
; *****  
; Author: 罗云彬  
; Web:      http://www.win32asm.com.cn （罗云彬的编程乐园）  
; E-mail: luoyunbin@sina.com  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
; Version 1.0  
;  
;          Date: 2004.05.01  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
; Sample.dll 导出函数：  
;  
; 1、 invoke _IncCounter  
;       增加 Dll 内部计数器的值（最大增加到 10）并返回计数值  
; 2、 invoke _DecCounter  
;       减少 Dll 内部计数器的值（最小减少到 0）并返回计数值  
; 3、 invoke _Mod,dwNumber1,dwNumber2  
;       输入： dwNumber1 和 dwNumber2 为两个整数  
;       输出： 两数的模 dwNumber1 % dwNumber2  
; *****  
  
_IncCounter    proto  
_DecCounter    proto  
_Mod           proto dwNumber1:dword, dwNumber2:dword
```

文件中不但列出了三个导出函数的函数声明，而且以注释的形式给出了函数的功能、调用方法和参数说明。（读者现在应该知道 MASM32 SDK 软件包中包含的 lib 文件和 inc 文件是怎么来的了！）

要是编写的动态链接库是供其他 C 程序员做开发用的，那么还应该书写 .h 头文件，.h 头文件的写法具体见 11.1.5 节。

虽然在前面的学习中一直在使用动态链接库，但本节仍然要介绍一下使用动态链接库的方法，这是为了比较全面地介绍使用动态链接库的不同途径和它们之间的区别。相关的例子文件包含在所附光盘的 Chapter11\Dll\ MASM Sample 目录中。在开始分析例子之前，首先要把上一节中生成的相关文件拷贝到本目录中以便使用，它们是 Sample.dll, Sample.lib 和 Sample.inc 文件。

[illegible]



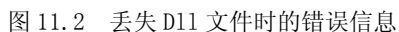
先看 UseDll11.asm 程序，这个程序用常规的方法实现了对 Sample.dll 动态链接库中函数的调用：

392

这段代码是再简单不过的了，读者只需要注意程序中用**黑体**标出的地方。

include	Sample.inc
includelib	Sample.lib

那么,这种方法有什么缺点呢?先来做几个实验。首先,将 Sample.dll 文件删除,再执行 UseDll11.exe 文件,这时对话框并没有被显示,系统弹出如图 11.2 所示的错误提示框。



393

法被装入执行。

第二个实验：修改上一节中的 Sample.asm，将入口函数中的返回值改为 FALSE，也就是说模拟 dll 初始化失败的情况。修改完毕后重新编译链接生成新的 Sample.dll 文件，并将文件拷贝到 UseDll11.exe 所在目录后运行，系统将弹出如图 11.3 所示的错误提示。也就是说，任何一个 dll 文件因为初始化失败而无法装入时，可执行文件也是无法被装入执行的。



图 11.3 Dll 初始化失败时的错误信息

第三个实验：模拟软件升级或 dll 文件版本不对时的情况，这种情况在 Windows 系统中经常发生，因为当某些应用软件包被安装的时候，它可能会用自己附带的某个版本的 dll 文件替换掉 Windows 目录中已存在的 dll 文件，当程序卸载的时候，它有可能会根据备份恢复原来的版本，但更多的情况是根本没有恢复，经过多次安装和卸载不同的应用软件包后，最终的结果就是 Windows 系统目录中各个 dll 文件的版本参差不齐。不同版本的 dll 文件中可能增加了一些函数，也可能废弃了一些函数，有时其他使用这个 dll 文件的程序可能刚好用到不存在的函数，而这个函数在原来版本的 dll 文件中本来是存在的。

现在就修改程序来模拟这种情况，将 Sample.def 文件中的 _DecCounter 一行去掉，这样 dll 文件的导出表中就不会有这个函数，相当于函数不存在了，然后重新编译 dll 文件并将它拷贝到 UseDll11.exe 所在目录，执行 UseDll11.exe，这时系统显示的是如图 11.4 所示的错误提示，UseDll11.exe 程序仍然不能被正常装入执行。



图 11.4 找不到导出函数的错误信息

现在读者一定明白这个最熟悉不过的错误信息的由来了，通常对付这种莫名其妙的错误的最好方法就是重新安装 Windows，这将使所有 dll 文件的版本被重新安装为统一的版本，错误也就自然消失了。读者也可能会说：把出错的 dll 替换掉不就行了吗，为什么要整个重装呢？问题是你知道原来的版本应该是多少吗？

使用标准的方法调用动态链接库中的函数，在源代码被编译链接成 exe 文件时，链接器会根据导入库中的信息将使用的 dll 文件名和函数名存放在 exe 文件头的导入表中，这样 Windows 要执行文件的时候，会根据导入表中的 dll 列表寻找每个 dll 文件，并根据函数名在每个 dll 中寻找导出函数，如果这中间出现任何错误，如上面演示的 dll 文件丢失、dll 文件初始化失败或 dll 中的函数名无法找到等情况，应用程序都无法被装载执行。

2. 方法二：动态装入

方法一的优点就是使用方便,应用程序可以像使用自己内部的函数一样使用 DLL 中的函数,缺点也显而易见,如果装入 DLL 的过程中有任何错误,应用程序没有任何机会完成应变的措施,因为它根本没来得及被装入执行。

编程中有时候会有下面的需求:

- 程序需要使用系统中的保留函数。这个函数确实存在于动态链接库的导出表中，可以被其他程序引用，但是软件开发包提供的 lib 文件中并不包含它。
- 不同版本 Windows 中的函数集不同（如 Windows NT 中的很多与安全有关的函数在 Windows 9x 中不存在），同一版本 Windows 中不同版本 dll 文件的函数集也可能不同，程序需要根据函数是否存在做不同的处理。
- 程序使用的某些库并不重要（如仅用来显示程序版本的库），如果丢失这个库，程序希望能继续运行，而不是像上面演示的那样出现根本无法装入的情况。

对于这些需求, 解决的办法就是不能将动态链接库的导入信息保存在可执行文件的导入表中, 也就是说不要让 Windows 系统来做动态链接库的装入工作, 这些工作由应用程序自己的代码来完成。有 3 个函数可以用来完成这样的功能: LoadLibrary (装入动态链接库), FreeLibrary (释放动态链接库) 和 GetProcAddress (获取导出函数地址)。

例子 UseD112.asm 程序使用这种动态装入的方法来实现 UseD111 程序同样的功能，来看看 UseD112.asm 的内容：

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include        windows.inc
include        user32.inc
includelib     user32.lib
include        kernel32.inc
includelib     kernel32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Equ 等值定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN equ    1000
DLG_MAIN equ    1000
IDC_COUNT equ   1001
IDC_INC      equ    1002
IDC_DEC      equ    1003
IDC_NUM1 equ    1004
IDC_NUM2 equ    1005
IDC_MOD      equ    1006

_PROCVAR2 typedef proto :dword, :dword
_PROCVAR0 typedef proto
_PROCVAR2 typedef ptr PROCVAR2

```

396

下面分析这个程序与 UseDll11 程序的不同点。

首先，程序的开始不再需要 Sample.lib 文件和 Sample.inc 文件，少了对应的 include 和 includelib 语句，因为并不需要链接器去定位函数的位置。

第二是 .const 数据段中需要自己定义装入的库文件名和函数名：

szDll	db	'Sample.dll', 0	;装入的动态链接库名称
szIncCounter	db	'_IncCounter', 0	;装入的函数名

szDecCounter	db	'_DecCounter', 0	;装入的函数名
szMod	db	'_Mod', 0	;装入的函数名

这些信息原来是由链接器根据 lib 文件的信息写在可执行文件头的导入表中的, 既然现在由程序自己来装入库函数, 那么这些信息也就需要自己定义了。

第三是在使用库中的函数之前需要使用 LoadLibrary 将库装入, 并使用 GetProcAddress 函数得到函数的入口地址。程序中将这个步骤安排在对对话框初始化消息 WM_INITDIALOG 中完成, 读者也可以在使用函数前的任何地方完成。LoadLibrary 函数的使用方法是:

invoke	LoadLibrary, lpDllFileName
.if	eax
	mov hDllInstance, eax
.endif	

参数 lpDllFileName 指向需要装载的库文件名, 库文件名是一个以 NULL 结尾的字符串。函数按下列顺序在不同目录中查找指定的库文件: 当前目录、Windows 系统目录和 PATH 环境变量列出的目录。如果这些目录中存在同名的库文件, 那么先搜索到的库文件会被装入。

如果装载成功, 函数返回库文件的实例句柄, 装载失败则返回 NULL。返回的实例句柄需要被保存起来, 以后在获取库中的导出函数、装载库中的资源, 以及释放库的操作中都要用到它。

对于方法一中库文件丢失和库的入口函数返回 FALSE 告诉 Windows 初始化失败的情况, LoadLibrary 函数均返回 NULL, 这样程序就可以根据情况决定该怎么做, 程序可以显示一个提示信息并退出, 也可以不使用这个库文件而继续执行。UseDll2 程序的处理方法是显示一个“Sample.dll 文件丢失或装载失败, 程序功能无法实现”提示信息, 然后将对话框中的“增加”、“减少”两个按钮, 以及输入的编辑框灰化后继续执行。这样, 对话框可以正常显示出来, 但是使用库函数的功能被屏蔽掉了。

如果装载动态链接库成功, 下一步就是使用 GetProcAddress 函数来获取库中函数的地址。GetProcAddress 函数的使用方法是:

invoke	GetProcAddress, hDllInstance, lpProcName
.if	eax
	mov lpProc, eax
.endif	

hDllInstance 参数就是 LoadLibrary 函数返回的动态链接库的实例句柄, lpProcName 指向要获取的函数名称, 函数名也是用以 NULL 结尾的字符串来定义。有些系统 DLL 中的函数名称并不是字符串, 而是使用数值编号, 对于这种情况, lpProcName 参数可以指定为函数的编号数值 (详见第 17 章的导出表部分)。

如果执行成功, 返回值是要获取的函数的入口地址, 程序可以保存它并在以后调用。如果执行失败, 比如, 因为版本变化等原因导致需要获取的函数不存在, 这时函数返回 NULL。

在不再需要动态链接库的时候, 为了释放库所占用的系统资源, 需要使用 FreeLibrary 函数释放它。FreeLibrary 函数的用法是:

invoke	FreeLibrary, hDllInstance
--------	---------------------------

输入参数是 LoadLibrary 函数返回的实例句柄，函数导致系统以 DLL_PROCESS_DETACH 代码调用库的入口函数，这样库文件可以自己释放占用的一些资源，然后，整个库的代码和数据被从应用程序的地址空间中清除。

但是在一个应用程序中使用 FreeLibrary 函数并不会影响另一个应用程序使用同一个库文件，当库文件还在被另一个程序使用的时候，它还是在物理内存中存在，操作系统为每个库文件维护一个装入计数器，每次使用 LoadLibrary 装载库文件（或者使用第一种方法由 Windows 来装入一个库）的时候，计数器递增；每次使用 FreeLibrary 函数将库释放的时候，计数器递减，只有到计数器减到零，也就是库文件没有被任何程序使用的时候，操作系统才会将它从物理内存中真正释放掉，否则仅是从某个进程的地址空间中解除了内存映射关系而已。

方法二和方法一的最后一个不同点是调用函数的方法，在使用 GetProcAddress 函数获取了库中导出函数的入口点以后，程序在调用的时候一般使用将参数手工入栈的方法，如对 _Mod 函数的调用可以写为：

push	num2	
push	num1	
call	lpMod	;lpMod 保存有 GetProcAddress 获取的地址

这样写法的缺点是无法使用 invoke 伪指令来进行参数检验，容易引发错误。实际上还有一个变通的方法，可以将一个变量定义为子程序入口指针，并为它定义参数个数，方法是两次使用 typedef 伪操作：

_PROCVAR2	typedef proto :dword, :dword
PROCVAR2	typedef ptr _PROCVAR2

如上面的第一句将 _PROCVAR2 类型定义为使用两个参数的函数类型，第二句将 PROCVAR2 类型定义为 _PROCVAR2 类型的指针，这样，在数据段中就可以将保存函数入口地址的变量使用 PROCVAR2 类型来定义了，得到的好处就是可以用 invoke 语句来调用这个变量中的指针：

	.data?	
lpMod	PROCVAR2	?

有人曾询问笔者这样一个问题：如果既没有导入库，也没有资料，该如何使用 DLL 中的函数？答案是：函数名是没有问题的，通过一些工具查看导出表就可以得知库中所有的导出函数列表，但是有关调用函数使用参数的数量和参数的定义方法等资料就成问题了，惟一的办法就是通过反汇编或者跟踪来找出参数的数量和含义后再通过本节介绍的方法调用。



使用方法二时要注意：不管是使用 LoadLibrary 函数还是 GetProcAddress 函数，对返回值必须要检查，否则一旦失败的话，很容易引发调用 NULL 指针的错误。

11.1.4 动态链接库中的数据共享

当多个应用程序同时使用同一个动态链接库的时候，这些动态链接库在系统中是存在于不同进程的地址空间中的，它们代表“宿主”程序工作，互相之间没有任何联系。这一点可以通过一个简单的实验来演示：当我们多次运行 UseDll1.exe（或 UseDll2.exe），按动不同对话框中的“增加”或“减少”按钮的时候，每个对话框中的计数值按照自己的规律增减，不会受到其他对话框中计数值的影响。这就是说，虽然 Sample.dll 被多个进程同时装入，但是操作系统为它们映射了各不相同的数据段，使它们工作起来互不影响。

但是需要在进程间进行数据共享的时候，这种互相隔离的特征就不是我们所需要的了，当然，解决的方法之一就是使用第 10 章中介绍的内存映射文件，但是更简单的办法是通过构造特殊的动态链接库来实现。

再次以前面的 Sample.dll 为例，现在将它的计数器改成是全局的，也就是说运行 UseDll1.exe 的多个拷贝的时候，不同对话框增减的是同一个计数器。

回顾第 2 章对 Link.exe 程序的介绍，会发现链接器有一个 /SECTION 选项，可以将某个节区的属性自行定义，选项中有一个 S 属性，代表将节区的属性设置为共享，这就是我们需要的，实际上不必修改 Sample.asm 源程序，只需要把 Makefile 文件中的 Link 选项修改一下：

```
DLL = Counter

ML_FLAG = /c /coff
LINK_FLAG = /subsystem:windows /Dll /section:.bss,S

$(DLL).dll: $(DLL).obj $(DLL).def
    Link $(LINK_FLAG) /Def:$(DLL).def $(DLL).obj
.asm.obj:
    ml $(ML_FLAG) $<
.rc.res:
    rc $<
```

未初始化数据段 .data? 的节区名称为 .bss，加上 /section:.bss,S 选项就可以将这个段的属性改为共享，这样，当 DLL 被不同应用程序装载的时候，不但映射到不同进程地址空间中的代码段来自同一段物理内存，.data? 段的映射也来自同一段物理内存。

修改 Makefile 以后来验证一下，使用 nmake /a 将 DLL 重新编译并将 Sample.dll 文件拷贝到 UseDll1 程序所在目录，然后多次执行 UseDll1 程序以产生多个对话框，当在一个对话框中按下“增加”按钮将计数增加到 x 的时候，再换到另一个对话框中再按“增加”按钮，会发现出现的值是 $x+1$ 而不是原来应该出现的 1，表示这些对话框操作的是一个共享的计数器。

如果不希望全部的数据都共享，如 hInstance 等私有的数据，可以把这些数据放在初始化数据段 .data 中，它的节区名称不同于 .data? 段，在将 .data? 段的属性修改为共享的时候并

不会影响 .data 段的属性。

11.1.5 在 VC++ 中使用动态链接库

在 Windows 下，动态链接库是混合编程的一种好方法，由于大部分语言支持动态链接库技术，所以将代码封装成动态链接库就可以在很广的范围内使用，常用的如在 C、C++ 或者 VB 中使用，不常用的如在 Oracle 数据库的存储过程中使用 Dll 中的函数，动态链接库远比在 DOS 使用静态库来进行混合编程要方便得多。

不同语言默认的调用约定和函数的命名方式是不同的，要想不同语言开发的动态链接库能够互相使用，链接库的开发语言和调用语言中的函数约定必须相同。语言对函数的约定有两种：调用约定和名字修饰约定。

调用约定决定了函数参数传送时入栈和出栈的顺序，以及堆栈平衡的方式（其细节在第 3 章的 3.4.2 节中就已经讲到了）。名字修饰约定指编译器在编译阶段如何定义函数的修饰名，各种编译器在编译函数时，会根据函数原型生成包含诸如函数名称、参数和返回值等信息的标识字符串，该字符串就称为函数的修饰名。

由于函数的修饰名仅在编译和链接阶段使用，所以该名称字符串存在于导入库 (*.lib) 文件中，而在最终生成的 Dll 文件中，导出的函数名和源文件中定义的函数名是一致的。

如果调用程序开发时和开发 Dll 使用的名字修饰约定不一致，那么在使用 Dll 对应的 lib 文件时，即使使用了正确的函数名，编译器仍然会报“找不到函数”错误，因为这时编译器在 lib 文件中找的是修饰名而不是函数名。

1. 各种语言的名字修饰约定

Win32 汇编语言使用的函数名字修饰是怎样的呢？我们可以从一个简单的例子看出来，首先观察下面的代码：

```

.586
.model flat,stdcall
option casemap:none

Test0
Test1
Test2
TestC
TestPascal

    proto
    proto    :dword
    proto    :dword, :dword
    proto    C :dword
    proto    PASCAL :dword

    .code

start:

    invoke   Test0
    invoke   Test1, 1
    invoke   Test2, 1, 2
    invoke   TestC, 1
    invoke   TestPascal, 1
    ret
end         start

```

由于这段代码仅仅对函数进行了声明和调用，实际上并没有一个动态链接库来提供这些函

数，所以链接的时候，链接器会报找不到外部函数符号的错误，从链接器报出的函数名就可以看出函数在 lib 文件中的修饰名应该是什么样子的：

```
Test.obj : error LNK2001: unresolved external symbol _Test0@0
Test.obj : error LNK2001: unresolved external symbol _Test1@4
Test.obj : error LNK2001: unresolved external symbol _Test2@8
Test.obj : error LNK2001: unresolved external symbol _TestC
Test.obj : error LNK2001: unresolved external symbol TESTPASCAL
Test.exe : fatal error LNK1120: 6 unresolved externals
```

Win32 汇编和 VC 使用的名字修饰方式是一样的，如果用 cl 编译链接下面的 C 代码，得到的错误提示是相同的（VC++ 从 5.0 版开始取消了对 PASCAL 约定的支持，所以演示代码中去掉了 TestPascal 函数）：

```
(1)      /* Test.c ---- 用 cl Test.c 编译 */
(2)      __stdcall Test0();
(3)      __stdcall Test1(int p1);
(4)      __stdcall Test2(int p1, int p2);
(5)      // __stdcall Test2(int p1, int p2, int p3);
(6)      TestC(int p1);
(7)      main ()
(8)      {
(9)          Test0();
(10)         Test1(1);
(11)         Test2(1, 2);
(12)         // Test2(1, 2, 3);
(13)         TestC(1);
(14)     }
```

在 VC 和 Win32 汇编中使用的名字修饰约定如下：stdcall 调用约定在输出函数名前加上一个下划线前缀，后面加上一个“@”符号以及参数的字节数，格式为“_函数名@参数字节数”；C 调用约定仅在输出函数名前加上一个下划线前缀，格式为“_函数名”。这可以从上面例子中链接器的输出信息中得到验证，另外，PASCAL 调用除了将函数名转为大写外无其他修饰。

由于 VC 和 Win32 汇编使用的名字修饰约定是相同的，所以在 VC 中使用 Win32 汇编写的 Dll 函数非常方便，只要在头文件中加上类似于例子中 __stdcall Test0() 方式的声明，然后直接进行调用就可以了。反过来，在 VC 写的 Dll 中，只要将函数定义为 stdcall 方式，那么在 Win32 汇编中就可以和使用系统 API 同样的方法来调用这些 Dll 函数。

但是，VC++ 使用的名字修饰约定却完全不同，将上面的 C 例子代码中第 5 和 12 行的注释符号去掉，并将文件存盘，改名为 Test.cpp，然后用命令 cl Test.cpp 进行编译，由于文件名有 cpp 后缀，VC 将使用 C++ 方式进行编译，这时得到的信息如下：

```
unresolved external symbol "int __cdecl TestC(int)" (?TestC@@YAHH@Z)
unresolved external symbol "int __stdcall Test2(int, int, int)" (?Test 2@@YGHHH@Z)
unresolved external symbol "int __stdcall Test2(int, int)" (?Test2@@YGH@Z)
unresolved external symbol "int __stdcall Test1(int)" (?Test1@@YGH@Z)
unresolved external symbol "int __stdcall Test0(void)" (?Test0@@YGHXZ)
```

可以看到, C++编译器使用的函数名称修饰方式比较复杂, 对于 `stdcall` 方式的约定, 编译器在函数名前面加 “?” 作为开始, 然后后面跟 “@@YG” 表示参数表开始, 参数表以下列代号表示: X 表示 `void`, D 表示 `char`, E 表示 `unsigned char`, F 表示 `short`, H 表示 `int`, I 表示 `unsigned int`……参数表的第一项为该函数的返回值类型, 其后依次为参数的数据类型, 参数表后以 “@Z” 标识整个名字的结束, 如果该函数无参数, 则以 “Z” 标识结束。

如例子中的 `Test1` 函数的返回值是 `int` 型的, 有一个 `int` 型的参数, 那么 `?Test1@@YG` 后面的第一个 H 表示返回值, 第二个 H 表示参数, 然后以 @Z 结束; 同理, `Test0` 函数修饰名中的 X 表示没有参数 (即 `void`), 这时修饰名以 Z 而不是 @Z 结束。

对于 C 方式, 函数名称修饰方式同上面的 `stdcall` 方式, 只是参数表的开始标识由上面的 “@@YG” 变为 “@@YA”。这一点可以从 `TestC` 函数的修饰名可以看出。

C++使用这样复杂的名称修饰方式的好处是可以支持重载等特征, 如例子中定义的两个 `Test2` 函数虽然函数名是相同的, 但是由于参数不同, 它们在 C++内部的修饰名是不同的, 这样编译器就可以从调用参数中区分究竟要去调用哪个函数。坏处就是 C++的名称修饰方式无法和其他语言兼容, 造成用 C++修饰方式命名的 Dll 函数无法被其他语言所使用。

幸亏 C++中还提供了 `extern "C"` 关键字, 在函数名和函数声明前加上 `extern "C"` 关键字后, C++对该函数强制使用标准 C 的函数名称修饰方式, 所以, 在 Dll 中用这种方式输出的函数可以被其他语言使用; 反过来, 其他语言编写的 Dll 函数在 C++的头文件中声明时, 前面也必须加上 `extern "C"` 关键字, 这样 C++才会在 lib 文件中找到正确的函数修饰名。

2. 书写供 C 语言使用的头文件

从上一段的介绍中可以总结出, 在写 C 语言头文件时, 如果头文件供 C 语言使用, 那么在函数声明前加 `__stdcall` 关键字, 如果供 C++语言使用, 那么函数声明前要加 `extern "C"` `__stdcall` 关键字。

为了使头文件可以同时使用在 C 和 C++中使用, 可以利用 C 编译器的内部变量 `__cplusplus` 进行条件编译, 当使用 C++方式编译时, 该变量将被设置成 `TRUE`, 否则被设置成 `FALSE`, 以前面 `Sample.dll` 例子中输出的函数为例, 改进后的头文件如下:

```
#ifdef __cplusplus
extern "C" {
#endif

__stdcall _IncCounter();
__stdcall _DecCounter();
__stdcall _Mod(unsigned num1, unsigned num2);

#ifdef __cplusplus
}
#endif
```

这样, 在 C++中使用时, 函数声明前就会自动被加上 `extern "C"` 关键字。

完整的在 VC++中使用 `Sample.dll` 的代码例子见光盘的 `Chapter11\Dll\VC++ Sample` 目录,

读者可以尝试将例子中的 `Mod.cpp` 文件改名为 `Mod.c` 并重新进行编译，就可以发现 `Sample.h` 和 `Sample.dll` 文件仍然可以正常使用。

11.2 Windows 钩子

11.2.1 什么是 Windows 钩子

1. Windows 钩子简介

在 DOS 操作系统下编程的时候，如果想截获某种系统功能，可以采取截获中断的办法。比如，要获取击键的动作可以截获 9 号中断，要获取应用程序对文件操作功能的调用可以截获 21h 号 DOS 中断，由于 DOS 是单任务系统，所以这些操作几乎全部是内存驻留程序做的。DOS 下截获中断的方法是这样的简单和随意，不管在驱动程序层次还是在应用程序层次都可以完成，以至于到最后截获操作被病毒泛滥成灾地使用。

在 Windows 下就不同了，我们已经知道保护模式下的中断描述符表是受系统保护的，在应用程序层次不可能再通过修改中断向量来截获系统中断了，但这样也对一些应用造成了不便，作为一种变通的措施，Windows 提供了钩子来完成类似的功能。那么，钩子是什么呢？Win32 API 手册中是这样描述的：

“A hook is a point in the Microsoft Windows message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure.”

翻译过来就是：“钩子是 Windows 的消息处理机制中的一个监视点，应用程序可以在这里安装一个监视子程序，这样就可以在系统中的消息流到达目的窗口过程前监控它们。”

也就是说，钩子可以用来截获系统中的消息流，显然，钩子不是像截获中断一样用来随心所欲地截获系统底层功能的，那么钩子能够用来做什么事情呢？（我仿佛听到了一些阴险的笑声……）不用笑得这么阴险嘛！大家想得没错，如果把钩子用在后台执行的程序中，就能够偷偷检查任何程序中发生的 `WM_CHAR` 消息，这样用户输入的任何内容：账号、密码、情书——不管是什么，不管是否显示在屏幕上——都可以被记录下来。事实上，很多木马程序就是这样做的，像冰河一类的木马程序就可以在后台记录用户的击键并偷偷发送到人家的信箱中去。

2. 钩子的类型

钩子是 Windows 消息机制中的监视点，应用程序可以在这里安装一个监视函数，这样就可以捕捉自己进程或者其他进程发生的事件。通过 `SetWindowsHookEx` 函数就可以做到这一点。`SetWindowsHookEx` 函数定义了监视函数的位置和监视消息的类型，这样，每当发生我们感兴趣的消息时，Windows 就会将消息发送给监视函数，监视函数是一个处理消息的回调函数，也称为“钩子函数”。

Windows 安装的钩子有两种类型：局部的和远程的。它们处理消息的范围不同。局部钩子仅钩挂属于自身进程的事件；远程钩子除了可以钩挂自身进程的事件，还可以钩挂其他进程中

发生的事件。远程钩子又分两种：基于线程的和系统范围的。基于线程的远程钩子用来捕获其他进程中某一特定线程的事件；而系统范围的远程钩子将捕捉系统中所有进程中发生的事件消息。

安装钩子会影响系统的性能，因为系统在处理所有的相关事件时都会调用钩子函数，特别是监视范围是整个系统范围的全局钩子。如果钩子函数中的处理代码过多的话，系统运行速度将会明显减慢，所以对于全局钩子一定要小心使用，不需要的时候应该立刻卸载。在 DOS 操作系统下编写中断服务程序的时候，如果代码有错误的话会影响其他调用它的程序。同样道理，由于钩子函数在其他进程的消息处理流程中插了一腿，所以一旦钩子函数存在问题的话，也会影响其他进程的运行。

可以把钩子想像成钓鱼钩，不同鱼钩用来钓的鱼是不同的，大钩钓大鱼，小钩钓小鱼，不同钩子钓的消息也是不同的，没有必要每次钓来所有的消息，根据监视的消息类型和时机的不同，钩子可以分为如表 11.1 所示的几种。

表 11.1 钩子的类型

钩子名称	监视消息的类型和时机
WH_CALLWNDPROC	每当调用 SendMessage 函数时，函数将消息发送给目标窗口过程前首先调用钩子函数
WH_CALLWNDPROCRET	每当调用 SendMessage 函数时，函数将消息发送给目标窗口过程后再调用钩子函数
WH_GETMESSAGE	每当调用 GetMessage 或 PeekMessage 函数时，函数从程序的消息队列中获取一个消息后调用钩子函数
WH_KEYBOARD	每当调用 GetMessage 或 PeekMessage 函数时，如果从消息队列中得到的是 WM_KEYUP 或 WM_KEYDOWN 消息，则调用钩子函数
WH_MOUSE	每当调用 GetMessage 或 PeekMessage 函数时，如果从消息队列中得到的是鼠标消息，则调用钩子函数
WH_HARDWARE	每当调用 GetMessage 或 PeekMessage 函数时，如果从消息队列中得到的是非鼠标和键盘消息，则调用钩子函数
WH_MSGFILTER	当用户对对话框、菜单和滚动条有所操作时，系统在发送对应的消息之前调用钩子函数，这种钩子只能是局部的
WH_SYSMSGFILTER	同 WH_MSGFILTER，不过是系统范围的
WH_SHELL	当 Windows shell 程序准备接收一些通知事件前调用钩子函数，如 shell 被激活和重画等
WH_DEBUG	用来给其他钩子函数除错
WH_CBT	当基于计算机的训练（CBT）事件发生时调用钩子函数
WH_JOURNALRECORD	日志记录钩子，用来记录发送给系统消息队列的所有消息
WH_JOURNALPLAYBACK	日志回放钩子，用来回放日志记录钩子记录的系统事件
WH_FOREGROUNDIDLE	系统空闲钩子，当系统空闲的时候调用钩子函数，这样就可以在这里安排一些优先级很低的任务

在这些钩子中，有些只能当做局部钩子使用，如 WH_MSGFILTER 钩子；有些只能当做系统范围的远程钩子使用，如 WH_JOURNALRECORD 和 WH_JOURNALPLAYBACK 钩子；而大多数的钩子可以在任何范围内使用。

对于不同的钩子，由于它们处理的消息类型不同，所以钩子函数的参数定义也是不同的，

在具体的编程中，需要查看 Win32 API 手册来了解各种钩子函数的参数定义。

另外，远程钩子和局部钩子的程序结构也是不同的。当安装了一个局部钩子时，每当指定的事件发生，Windows 就可以调用进程中的钩子函数；但是若安装的是远程钩子，系统不能从其他进程的地址空间中调用钩子函数，因为两个进程的地址空间是隔离的，又由于系统中只有 DLL 程序是可以插入到其他进程的地址空间中去的，所以远程钩子的钩子函数必须位于一个动态链接库中，而且必须是共享数据段的动态链接库（因为写远程钩子要用到动态链接库，所以本书中将两部分内容合在一章中介绍）。

但是也有两个例外：日志记录钩子和日志回放钩子虽然属于远程钩子，但是它们的钩子函数却可以放在安装钩子的程序中，并不需要单独放在一个动态链接库中。Microsoft 并没有说明为什么有这样的例外，笔者认为其中的原因是这两个钩子是用来监控比较底层的硬件事件的，所以钩子函数的调用并不是从其他进程的地址空间中发起的，而是从 Windows 内部发起的，所以不存在不同进程之间地址空间隔离的问题（猜想而已，如果读者有明确的资料请告知笔者）。

下面的 11.2.2 节以键盘钩子为例来说明系统范围远程钩子的安装和使用，局部钩子的使用步骤与之类似，只不过不必将钩子函数放在动态链接库中而已，使用起来更加简单，读者可以举一反三自己尝试一下。11.2.3 节中演示日志钩子的使用方法。

11.2.2 远程钩子的安装和使用

1. 钩子程序的结构

钩子程序一般包括 3 个功能模块：

- （1）主程序——用来实现界面或者其他功能。
- （2）钩子回调函数——用来接收系统发过来的消息。
- （3）钩子的安装和卸载模块。

对于局部钩子来说，这些模块可以处在同一个可执行文件中。而对于远程钩子来说，第 2 部分必须放在一个动态链接库中，第 3 部分虽然没有要求，但一般也放在动态链接库中，这是因为钩子创建以后得到一个钩子句柄，这个句柄要在钩子回调函数中，以及卸载钩子的时候用到，如果把这部分代码放在主程序中的话，还需要创建一个函数将它传回给动态链接库，所以还不如直接放到库中。

所附光盘的 Chapter11\KeyHook 目录中的例子采用的就是这样的结构。目录中包括两部分文件：HookDll.asm 和 HookDll.def 文件用来生成动态链接库；Main.asm 和 Main.rc 是主程序部分。程序用一个系统范围的远程钩子来实现监视所有键盘输入的功能。由于安装钩子回调函数的动态链接库要求是共享数据段的，所以请读者注意 Makefile 中 dll 文件的链接选项，它使用了 /section:.bss,S 选项。Makefile 文件的内容如下：

```
NAME = Main
DLL = Hookdll

ML_FLAG = /c /coff
```

HookDll.asm 文件的内容如下:

407

需要共享的变量被放在 `.data?` 段中，如钩子句柄和钩住的按键内容等，仅 `dll` 程序的实例句柄不需要共享，不需要共享的变量放在 `.data` 段中。动态链接库的入口函数例行公事地返回了一个 `TRUE` 来表示允许被装入。程序中只写了 3 个函数，`HookProc` 是钩子回调函数，`InstallHook` 和 `UninstallHook` 函数是供主程序使用的钩子安装函数和卸载函数。由于这 3 个函数是需要导出的，所以 `HookDll.def` 文件中包括了它们的名称：

哭

```

invoke SetWindowsHookEx, idHook, lpHookProc, hInstance, dwThreadId
.if eax
mov     hHook, eax
.endif

```

[illegible]

[illegible]

为了使用动态链接库中的导出函数 `InstallHook` 和 `UninstallHook`，在程序的开头需要用 `include` 语句和 `includelib` 语句将动态链接库的函数声明和导入库包含进来。

在平时，主程序等待自定义消息 WM_HOOK，并将传递过来的按键字符串通过发送 EM_REPLACESEL 消息添加到编辑框中，在添加之前先检测按键是否为回车键，如果是，再人为地插入一个换行符 (0ah)，以便将编辑框中的内容换行显示。

现在回过头来看 HookDll.asm 程序中的钩子回调函数，回调函数的写法一般如下：

各种类型钩子的回调函数的参数都是这样 3 个，但是它们的定义各不相同，就像窗口过程在收到各种不同消息的时候，wParam 和 lParam 的定义各不相同一样，不同类型的钩子回调函数的返回值定义也是各不相同的。

- dwCode——键盘消息的处理方式。如果是 HC_ACTION，表示收到一个正常的击键消息；如果是 HC_NOREMOVE，表示对应消息并没有从消息队列中移去（当某个进程用指定 PM_NOREMOVE 标志的 PeekMessage 函数获取消息时就是如此）。
- wParam——按键的虚拟码（即 Windows.inc 中定义的 VK xxx 值）。

- lParam——按键的重复次数、扫描码和标志等数据，不同数据位的定义如下：
 - 位 0~15：按键的重复次数。
 - 位 16~23：按键的扫描码。
 - 位 24：按键是否是扩展键（F1 与 F2 等 Fx 键，小键盘数字键等），如果此位是 1 表示按键是扩展键。
 - 位 25~28：未定义。
 - 位 29：如果 Alt 键在按下状态，此位置 1，否则置 0。
 - 位 30：按键的原先状态，消息发送前按键原来是按下的，此位被设置为 1，否则置 0。
 - 位 31：按键的当前动作，如果是按键按下，那么此位被设置为 0；按键释放的话被设置为 1。

对于每个击键动作，钩子回调函数会在键按下和释放的时候被调用两次，只需根据 lParam 的位 31 中的标志来记录一次，否则得到的是重复信息。

另外，回调函数收到的参数是以按键的扫描码和虚拟码表示的，在送给主窗口前需要将它转换成我们认识的 ASCII 码，但虚拟码或扫描码和 ASCII 码之间的对应关系并没有规律，必须进行查表操作才能转换。如果在程序中自己转换的话，需要一个键码对应表和查表程序。

Windows 中现成的函数 ToAscii 可以完成这个功能并自动辨认按键的按下或释放动作。代码如下。

HookProc	proc	_dwCode, _wParam, _lParam
	local	@szKeyState[256]:byte
	invoke	CallNextHookEx, hHook, _dwCode, _wParam, _lParam
	invoke	GetKeyboardState, addr @szKeyState
	invoke	GetKeyState, VK_SHIFT
	mov	@szKeyState + VK_SHIFT, al
	mov	ecx, _lParam
	shr	ecx, 16
	invoke	ToAscii, _wParam, ecx, addr @szKeyState, addr szAscii, 0
	mov	byte ptr szAscii [eax], 0
	invoke	SendMessage, hWnd, dwMessage, dword ptr szAscii, NULL
	...	

ToAscii 函数的用法是：

invoke	ToAscii, dwVirtKey, uScanCode, lpKeyState, lpBuffer, uFlags
--------	---

dwVirtKey 参数指定按键的虚拟码，在使用时直接用钩子回调函数的 wParam 参数就可以了，uScanCode 指定按键的扫描码，并用位 15 来表示是按键按下还是按键释放，与回调函数的 lParam 参数对比可以看出，lParam 参数的高 16 位就是需要的数据，所以程序将 lParam 右移 16 位后用做 uScanCode 参数。

lpKeyState 指向一个 256 字节的缓冲区，其中存放键盘中所有按键的当前状态，一个字节

表示一个按键，数值为 1 表示按下，为 0 表示释放，数据在缓冲区中的排列位置按照 VK_XX 虚拟码的顺序排列。这是为了让函数得知键盘上各种控制键的状态（如 Shift, Alt 和 Ctrl 等），因为这些键是否按下对转换结果是有影响的，比如，同样是按键“1”，如果 Shift 键不按，对应的就是“1”，按下的话函数必须返回“!”才是正确的结果。当然不可能自己去填写这个缓存区，使用 GetKeyboardState 函数就可以让系统根据当前的键盘状态填写这个缓冲区。

lpBuffer 指向一个缓存区，用来接收转换后的 ASCII 码，最后的 uFlags 参数表示当前是否有一个菜单在激活状态，0 表示没有，1 表示有菜单正在激活。

函数的返回值表示转换后返回在 lpBuffer 缓冲区中的字符数量，它可能是 0（如按键放开时不产生字符）、1 或者是 2，下面的语句根据返回字符数将缓冲区中的字符尾部加上一个 NULL：

```
mov     byte ptr szAscii [eax],0
```

对于 Shift 等控制键来说，GetKeyboardState 函数返回的状态是区分左、右键的（分别对应 VK_LSHIFT 和 VK_RSHIFT），而 ToAscii 函数检测的是 VK_SHIFT，不对 Shift 键进行处理的话，转换结果可能是错误的，所以程序使用 GetKeyState 函数单独获取 VK_SHIFT 的状态并手工修改缓冲区中 VK_SHIFT 位置的状态。

转换完成后，用 SendMessage 函数将转换后的按键内容传递给主窗口，就大功告成了！不过要注意的还有两点：如果钩取的不是键盘消息而是其他窗口消息，在这里就应该使用 PostMessage 而不是 SendMessage 函数，否则可能造成死循环；其次不要向主窗口传递地址，因为钩子 DLL 被插入到其他进程的地址空间中运行，所以将地址传回去可能是无效的。

不同类型钩子回调函数返回值的定义是不同的。对于键盘钩子，返回 0 表示允许 Windows 将消息转发给目标窗口过程，返回非 0 值表示让 Windows 将消息丢弃，这样钩子函数可以检测到按键动作，目标程序却无法收到键盘消息，相当于所有的按键都失效了。



需要再次提醒的是：不同类型钩子的回调函数的参数，以及返回值的含义都是不同的，上面的例子是键盘钩子的情况，在使用其他类型钩子的时候请参考手册中对参数和返回值的说明。曾经有读者将上面键盘钩子的例子几乎原封不动地套到其他钩子的使用中，结果总是无法调试成功，其实原因就在于此。

3. 钩子链

Windows 系统中可以同时存在多个同类型的钩子，多个程序同时安装同一种钩子的时候就会出现这种情况，这些钩子组成一个钩子链，最近加入的钩子放在链表的头部，Windows 负责为每种钩子维护一个钩子链。当一个事件发生的时候，Windows 调用最后安装的钩子，然后由当前钩子的回调函数发起调用下一个钩子的动作，Windows 收到这个动作后，再从链表中取出下一个钩子的地址并将调用传递下去。

在大多数的情况下，一个钩子回调函数最好把消息事件传递下去以便其他的钩子都有获得处理这一消息的机会。调用下一个钩子函数是 CallNextHookEx，该函数的用法是：

```
invoke CallNextHookEx, hHook, dwCode, wParam, lParam
```

hHook 参数是当前钩子的句柄，dwCode, wParam 和 lParam 参数就是当前钩子收到的参数，这个函数让 Windows 调用钩子链中的下一个钩子。如果调用成功，函数的返回值是下一个钩子回调函数返回的数值。

11.2.3 日志记录钩子

日志记录钩子是一种特殊的钩子，说它特殊是因为它是远程钩子，却不用放在动态链接库中，这就为监视系统范围的消息提供了方便。本节中尝试用日志记录钩子的办法来实现键盘监视的功能，源代码包括在所附光盘的 Chapter11\RecHook 目录中，包括汇编源文件 RecHook.asm 和资源脚本文件 RecHook.rc，其中 RecHook.rc 文件的内容和上一个例子的 Main.rc 文件是一样的。

RecHook.asm 文件的内容如下:

[illegible]

```
HookProc endp  
:  
:
```

415

```

        .else
            invoke    EndDialog, hWnd, NULL
        .endif
;*****
        .else
            mov       eax, FALSE
            ret
        .endif
        mov       eax, TRUE
        ret

_ProcDlgMain    endp
;>>>>>>>>>
start:
        invoke     GetModuleHandle, NULL
        mov        hInstance, eax
        invoke     DialogBoxParam, eax, DLG_MAIN, NULL, \
            offset _ProcDlgMain, NULL
        invoke     ExitProcess, NULL
;>>>>>>>>>
        end        start

```

由于不再需要动态链接库了，钩子回调函数 HookProc 被移到了主程序中，也取消了 InstallHook 和 UninstallHook 两个子程序，相应的内容直接放在 WM_INITDIALOG 和 WM_CLOSE 消息中完成。在 WM_INITDIALOG 消息中用下面的语句完成对钩子的安装：

```
invoke    SetWindowsHookEx, WH_JOURNALRECORD, addr HookProc, hInstance, NULL
```

参数 WH_JOURNALRECORD 表示安装的钩子是日志记录钩子。

由于钩子回调函数也写在主程序中，所以没有必要再通过自定义的 WM_HOOK 消息来通信，在回调函数中使用 ToAscii 函数将监测到的按键扫描码转换成 ASCII 码字符串以后，程序直接发送 EM_REPLACESEL 消息将它添加到编辑框中。读者可以和前面一个例子对比一下这些细节上的不同。

程序比较重要的一个不同点在于日志钩子回调函数的参数定义不同，在这里 dwCode 的参数定义如下：

- HC_ACTION——系统准备从消息队列中移去一条消息，消息的具体信息由 lParam 参数中指定的 EVENTMSG 结构定义。
- HC_SYSMODALOFF——某个系统模态对话框准备被关闭。
- HC_SYSMODALON——某个系统模态对话框准备被建立。

我们关心的是 HC_ACTION 标志，当发生 HC_ACTION 标志的消息时，lParam 参数指向一个 EVENTMSG 结构，其定义为：

```
EVENTMSG STRUCT
```


message	DWORD	?	;消息队列中将要移去的消息 ID
paramL	DWORD	?	;消息的 wParam 参数
paramH	DWORD	?	;消息的 lParam 参数
time	DWORD	?	;消息发生的事件
hwnd	DWORD	?	;消息对应的窗口句柄

EVENTMSG ENDS

由于日志记录钩子可以截获的不仅是键盘消息，也有鼠标等其他消息，所以需要有个地方指定消息类型，通过检测 EVENTMSG 结构中的消息 ID 字段就可以得知截获的究竟是什么消息。如果关心的是按键消息的话，那么发现消息 ID 为 WM_KEYDOWN 时进行处理就可以了，同理，如果关心的是鼠标消息的话，使用日志记录钩子也可以完成鼠标钩子完成的工作。例子程序中的相关代码如下：

```
.if      _dwCode == HC_ACTION
    mov     ebx, _lParam
    assume  ebx:ptr EVENTMSG
    .if     [ebx].message == WM_KEYDOWN
        ;处理按键消息
    .endif
    assume  ebx:nothing
.endif
```

日志记录钩子回调函数的返回值没有被定义。所以不管返回什么值对消息的传递都没有影响。

第 12 章

多线程

12.1 进程和线程





进程和线程在字面上看起来颇为相近，又因为两者是息息相关的，所以往往给初学者造成混淆，其实从英文原文来看，进程（Process）和线程（Thread）是完全不同的。在开始介绍本章中的多线程和下一章中有关进程的内容之前，首先在这里介绍进程和线程的概念，以及它们之间的联系。

进程是正在执行中的应用程序，磁盘上存储的可执行文件只能称之为文件而不能称为进程，内存中正在执行的文件才叫做进程。一个进程是一个执行中的文件使用资源的总和，包括虚拟地址空间、代码、数据、对象句柄、环境变量和执行单元等。当一个应用程序同时被多次执行时，产生的是多个进程，因为虽然它们由同一个文件执行而来，但是它们的地址空间等资源是互相隔离的，这与不同文件在执行的情况是一样的。

进程是不“活泼”的，要使进程中的代码被真正运行起来，它必须拥有在这个环境中运行代码的“执行单元”，这就是线程，线程是操作系统分配处理器时间的基本单位，一个线程可以看成是一个执行单元，它负责执行包含在进程地址空间中的代码。当一个进程被建立的时候，操作系统会自动为它建立一个线程，这个线程从程序指定的入口地址开始执行（对于 Win32 汇编，就是源代码最后 start 指定的入口地址），通常把这个线程称为主线程，当主线程执行完最后一句代码的时候，进程也就没有继续存在的理由了，这时操作系统会撤销进程拥有的地址空间和其他资源，对我们来说，这就意味着程序的终止。

在主线程中，程序可以继续建立多个线程来“同时”执行进程地址空间中的代码，这些线程被称为子线程。操作系统为每个线程保存单独的寄存器环境和单独的堆栈，但是它们共享进程的地址空间、对象句柄、代码和数据等其他资源，它们可以执行相同的代码，可以对相同的数据进行操作，也可以使用相同的句柄。读者可以把一个进程中的多个线程看成是进程范围内的“多任务”。

进程和线程的关系可以看做是“容器”和“内容物”的关系，进程是线程的容器，线程总是在某个进程的环境中被创建，它不可以脱离进程而单独存在，而且线程的整个生命周期都存



12.2 多线程编程

12.2.1 一个单线程的“问题程序”

来看一个“问题程序”，假设编写一个计数程序，程序的要求如下：

-

图 12.1 一个有问题的计数程序

程序的代码在所附光盘的 Chapter12\Counter 目录中，里面的 Counter.rc 文件定义了如图 12.1 所示的对话框，其代码如下：

[illegible]

汇编源文件写起来似乎很简单，只要在 WM_COMMAND 消息中对“计数”和“暂停/恢复”按钮的动作进行处理就可以了。在“计数”按钮中，可以调用一个计数子程序不停地进行加法运算并将结果显示出来，为了能够随时停止或暂停，可以设置一个标志位，按动“停止计数”或者“暂停/恢复”按钮时设置不同的标志，计数子程序在循环中通过测试这个标志位来决定是否暂停或退出。按照这个思路，程序可以写成目录中 Counter.asm 所示的样子：

420

421

```

;*****
        .elseif    eax ==    WM_INITDIALOG
            push    hWnd
            pop     hWinMain
            invoke   GetDlgItem, hWnd, IDOK
            mov     hWinCount, eax
            invoke   GetDlgItem, hWnd, IDC_PAUSE
            mov     hWinPause, eax
;*****
        .else
            mov     eax, FALSE
            ret
        .endif
        mov     eax, TRUE
        ret

_ProcDlgMain    endp
;>>>>>>>>>
start:
            invoke   GetModuleHandle, NULL
            mov     hInstance, eax
            invoke   DialogBoxParam, eax, DLG_MAIN, \
                NULL, offset _ProcDlgMain, NULL
            invoke   ExitProcess, NULL
;>>>>>>>>>
        end        start

```

这个程序很简单：当用户按下“计数”按钮的时候，WM_COMMAND 消息处理代码调用 _Counter 子程序进行计数，子程序会将 IDOK 按钮上的文字通过 SetWindowText 函数改为“停止计数”，并且使用 EnableWindow 函数激活“暂停/恢复”按钮，然后进入计数循环。

在循环中，程序通过 dwOption 变量中的第 1 位（预定义为 F_STOP）来判断是否停止，通过第 0 位（预定义为 F_PAUSE）来决定是否暂停计数。这些标志位的状态以后会在按下“停止计数”或“暂停/恢复”按钮时在 WM_COMMAND 消息中设置。

粗看起来，程序天衣无缝，现在运行一下看看——“计数”按钮被正确地改为“停止计数”，“暂停/恢复”按钮也正确地被激活了，但是接下来就不对了，编辑框中并没有显示计数值，更糟糕的是，接下来所有的按钮都无法按动，对话框也无法移动和关闭，总之，程序停止了响应，现在能结束它的惟一办法是通过任务管理器强制结束！

为什么会这样呢？这是因为主线程自从开始进入计数循环以后，就一直在那里“埋头苦干”，忙于计数工作，以至于把 WM_COMMAND 消息的处理抛到脑后去了，WM_COMMAND 消息没有返回，对话框内部的消息循环就停留在 DispatchMessage 函数里面，以至于消息队列中的后续消息堆积在那里无法处理，这样，不管用户按动“停止计数”按钮也好，移动对话框也好，这些动作虽然会被 Windows 检测到并转换成相应的消息放入消息循环中去，但是这些消息堆积在那里无法处理，所以就看不到对话框有任何的响应。

程序进入了一个怪圈：停止或暂停循环的条件是设置标志位，标志位是在按动“停止计数”或“暂停/恢复”按钮的 WM_COMMAND 消息中设置的，而 WM_COMMAND 消息被堆积在消息队列中

哭

423

424

[illegible]

将修改后的源程序与修改前的对比一下，可以发现不同的只有两个地方：第一是处理“计数”按钮的 WM_COMMAND 消息中，调用 _Counter 子程序的指令变成了对 CreateThread 函数的调用，这个函数就是用来创建新线程的函数；第二是 _Counter 子程序的定义有点不同，这是因为用做线程入口的线程函数必须按照规定的格式定义。这两个不同点其实也可以归结为一个，因为第二个不同点实际上是对第一个不同点的配合。

是不是这样修改后程序就正常工作了呢？可以来验证一下：运行程序，按下“计数”按钮，这次在“计数”按钮被改为“停止计数”，“暂停/恢复”按钮被激活的同时，计数值可以在编辑框中显示出来了，而且在计数的过程中，可以移动程序位置、关闭程序、按动各个按钮——总之，计数线程和主线程在同时工作了。

接下来参考这个程序来探讨多线程程序的结构。

2. 多线程程序的结构

Windows 中存在很多类型的对象：如窗口类、窗口、文件、菜单、图标、光标和钩子等，当一个线程创建某个对象的时候，这个对象归线程所属的进程所拥有，进程中的其他线程也可以使用它们，比如，可以在主线程中打开一个文件，然后在另一个子线程中读写这个文件。

大部分类型的对象不属于创建它的线程，而是属于进程，这表现在创建对象的线程结束时，如果线程不去主动删除这些对象，系统不会自动删除它们，只有当整个进程结束时对象还没有被删除，系统才会自动删除它们。但是窗口和钩子这两种对象比较特殊，它们首先是由创建窗口和安装钩子的线程所拥有的，如果一个线程创建一个窗口或安装一个钩子，然后线程结束，那么系统会自动摧毁窗口或卸载钩子。

进程中的消息队列则与线程和窗口都是相关的，如果在一个线程中创建了一个窗口，那么 Windows 就会单独给这个线程分配一个消息队列，为了让这个窗口工作正常，线程中就必须存在一个消息循环来派送消息，这就是主线程中只有当创建窗口时才需要消息循环代码，不创建窗口的程序（如控制台程序）就不需要消息循环的原因。这也就意味着如果窗口是在子线程中创建的，主线程中的消息循环根本不会获得这个窗口的消息，子线程必须自己设置一个消息循环。当使用 `SendMessage` 或者 `PostMessage` 函数向一个窗口发送消息的时候，系统会先判别窗口是被哪个线程创建的，然后把消息派送到正确线程的消息队列中。

整理一下思路：如果在一个线程中创建窗口，就必须设置消息循环，有了消息循环，就必须遵循 1/10 秒规则，这就意味着这个线程不该用来处理长时间的工作。而在一个程序中为不同的线程设置多个消息循环，不但使代码复杂化，而且会产生诸多的其他问题，所以在多线程程序中，规划好程序的结构是很重要的。

规划多线程程序的原则是：首先，处理用户界面（指拥有窗口和需要处理窗口消息）的线程不该处理 1/10 秒以上的工作；其次，处理长时间工作的线程不该拥有用户界面。根据这个规则，我们大致可以把线程分成两大类：

- 处理用户界面的线程——这些线程创建窗口并设置消息循环来负责处理窗口消息，一个进程中并不需要太多这种线程，一般让主线程负责这个工作就可以了。
- 工作线程——该类线程不处理窗口界面，当然也就不处理消息了，它一般在后台运行，干一些繁重的需要长时间运行的粗活。

一般来说，处理用户界面的线程交给主线程来做就可以了。如果主线程中接到一个用户指令，完成这个指令可能需要比较长的时间，主线程可以建立一个工作线程来完成这个工作，并负责指挥这个工作线程。这与我们日常生活中的许多例子是很像的，比如，公司的经理就好比是用户界面线程，他负责和外界沟通、谈判业务、对董事会（指对着屏幕单击鼠标的用户）汇报，同时负责将具体的工作分配给职能部门（也就是工作线程）做，如果让经理具体地去做每一件事情，下车间去包装产品或开着卡车外出拉原料，那么他就无法管理好这个公司了。

在多线程版本的 `Counter.asm` 例子中，使用的就是这样的程序结构：主线程用来维护界面，接收用户的输入动作并安排相应的操作，工作线程则用来进行计数操作。

3. 线程之间的通信

主线程在创建工作线程的时候，可以通过参数给工作线程传递初始化数据，当工作线程开始运行后，还需要通过通信机制来控制工作线程，就像经理虽然不用亲自干活，也需要随时了解和控制情况一样；同样，工作线程有时候也需要将一些情况主动通知主线程。

线程之间的通信可以归纳为 3 种方法。

使用全局变量传递数据是最常用的方法，如例子中主线程通过设置全局变量 dwOption 中的数据位来通知工作线程，工作线程随时检查这个变量并根据要求做相应的动作；反过来，工作线程也通过设置 dwOption 的第 2 位（预定义为 F_COUNTING）来控制主线程中对 IDOK 按钮的动作，如果 F_COUNTING 被置位，表示线程在运行中，这时 IDOK 按钮被定义为“停止计数”按钮，否则 IDOK 按钮被定义为“计数”按钮。使用全局变量传递数据的缺点是当多个工作线程使用同一个全局变量时，可能会引起同步问题，在 12.4 节中会探讨这个问题。

第 2 种方法是通过发送消息来通信，如工作线程工作结束时，可以通过向主线程发送自定义的 WM_XXX 消息来通知主线程，这样主线程就不需要随时去检查工作线程是否结束。只要在窗口过程中处理 WM_XXX 消息就可以了。这种方法的缺点是无法向工作线程发送消息，因为工作线程中一般并没有消息队列，所以这种方法仅用在工作线程向主线程传递消息的应用中。

如果线程之间传递的不是数据而是代表状态的布尔值，也可以使用第 3 种方法，即使用事件对象来通信，相关内容会在 12.3 节中详细介绍。

12.2.3 与线程有关的函数

1. 创建线程

创建一个线程可以使用 CreateThread 函数，函数的用法是：

invoke	CreateThread, lpThreadAttributes, dwStackSize, lpStartAddress, \
	dwParameter, dwCreationFlags, lpThreadId
.if	eax
	mov hThread, eax
.endif	

函数使用的参数定义如下：

- lpThreadAttributes——指向一个 SECURITY_ATTRIBUTES 结构，用来定义线程的安全属性，这个结构在 CreateFile 函数的介绍中已经涉及过，主要用来指定句柄是否可以继承，如果想让线程使用默认的安全属性，可以将参数设置为 NULL。
- dwStackSize——线程的堆栈大小。如果指定为 0，那么线程的堆栈大小和主线程使用的大小相同。系统自动在进程的地址空间中为每个新线程分配私有的堆栈空间，这些空间在线程结束的时候自动被系统释放，如果需要的话，堆栈空间会自动增长。
- lpStartAddress——线程开始执行的地址。这个地址是一个规定格式的函数的入口地址，这个函数就被称为“线程函数”。
- dwParameter——传递给线程函数的自定义参数。

- `dwCreationFlags`——创建标志。如果是 0，表示线程被创建后立即开始运行，如果指定 `CREATE_SUSPENDED` 标志，表示线程被创建后处于挂起状态，直到使用 `ResumeThread` 函数显式地启动线程为止。
- `lpThreadId`——指向一个双字变量，用来接收函数返回的线程 ID。线程 ID 在系统范围内是惟一的，一些函数需要用到线程 ID。

如果线程创建成功，函数返回一个线程句柄，这个句柄可以用在一些控制线程的函数中，如 `SuspendThread`、`ResumeThread` 和 `TerminateThread` 等函数，如果线程创建失败，那么函数返回 `NULL`。

当程序调用 `CreateThread` 函数时，首先系统为线程建立一个用来管理线程的数据结构，其中包含线程的一些统计信息，如引用计数和退出码等，这个数据结构被称为线程对象；接下来系统将从进程的地址空间中为线程的堆栈分配内存并开始线程的执行。当线程结束时，线程的堆栈被释放，但是线程对象不会马上被释放，系统保留它以便其他线程可以通过它检测线程的有关情况，直到使用 `CloseHandle` 函数关闭线程句柄后，线程对象才会被释放。



但是线程对象也可以提前被释放，对于大部分的句柄来说（如文件句柄 `hFile`，文件寻找句柄 `hFindFile` 等），使用 `CloseHandle` 函数关闭句柄意味着整个对象被释放，但对于线程句柄来说，关闭它仅释放线程的统计信息，并不会终止线程的执行，所以如果不再需要使用线程句柄的话，在调用 `CreateThread` 后马上就可以将它关闭掉，线程的执行并不会受影响。

2. 线程函数

如果创建线程时没有指定 `CREATE_SUSPENDED` 标志，当 `CreateThread` 函数返回时，`lpStartAddress` 参数指向的线程函数就已经开始运行了。线程函数包含所有需要在线程中执行的代码，它有一个输入参数，线程函数的一般书写格式是：

<code>_ProcThread</code>	<code>proc uses ebx esi edi lParam</code>
	<code>local 局部变量</code>
	<code>...</code>
	<code>mov eax, 返回码</code>
	<code>ret</code>
<code>_ProcThread</code>	<code>endp</code>

读者可以自由定义函数的名称，只要在使用 `CreateThread` 函数时将 `lpStartAddress` 参数指向函数的入口地址就可以了，`lParam` 参数传递过来的就是调用 `CreateThread` 函数时使用的 `dwParameter` 参数。

向线程函数传递参数的时候，读者可能会觉得一个 `lParam` 参数不太够用，如果需要传递多个参数该怎么办呢？其实这不是问题，因为子线程和主线程使用同一个地址空间，主线程可以通过全局变量来传递参数。

有时候也可能遇到这种情况：进程中存在多个子线程，这些子线程的线程函数使用同一个子程序，如果对这些子线程使用同样的全局变量传递参数，难免会引起冲突。这时可以为每个子线程分配一个存放参数的内存块，主线程通过 `lParam` 参数把内存块的指针传递给子线程，子线程通过这个指针存取内存块中的内容就可以了，不过在子线程结束的时候不要忘了释放内存块。

3. 终止线程

线程从线程函数的第一句代码开始执行，直到线程被终止为止。当线程被正常终止时，系统会进行下面的操作：

- 线程使用的堆栈被释放。
- 系统将线程对象中的退出代码设置为线程的退出码。
- 系统将递减线程对象的使用计数。

由于线程结束后的退出码可以被其他线程用 `GetExitCodeThread` 函数检测到，所以可以当做自定义的返回值来表示线程执行的结果。终止一个线程的执行有 4 种方法。

第 1 种方法是线程函数的自然退出，当函数执行到一句 `ret` 指令返回时，Windows 将终止线程的执行，这时放在 `eax` 中的返回值就是线程的退出码。一般建议使用这种方法终止一个线程的执行。

第 2 种方法是使用 `ExitThread` 函数来终止线程：

invoke	ExitThread, dwExitCode
--------	------------------------

`ExitThread` 函数只能用于终止当前线程，它并不能用于在一个线程中终止另外一个线程，和 `ExitProcess` 函数一样，`ExitThread` 函数不会有返回的时候。`dwExitCode` 参数指定为线程的退出码。使用 `ExitThread` 函数和使用 `ret` 指令终止线程的效果是一样的，但显然不如使用 `ret` 指令来得简洁和方便。

第 3 种方法是使用 `TerminateThread` 函数，这个函数可以用来在一个线程中强制终止另一个线程的执行：

invoke	TerminateThread, hThread, dwExitCode
--------	--------------------------------------

`hThread` 参数指定需要终止的线程句柄，`dwExitCode` 将用做被终止线程的退出码。如果函数执行成功，返回值是非 0 值，否则函数返回 0，但是 `TerminateThread` 函数是一个异步执行的函数，即使函数返回非 0 值，也并不代表目标线程已经终止，可能终止的过程还要延续一段时间，如果必须确认线程已经真正结束的话，可以使用 `GetExitCodeThread` 函数来检测。

`TerminateThread` 函数是一个被强烈建议避免使用的函数，因为一旦执行这个函数，程序无法预测目标线程会在何处被终止，其结果就是目标线程可能根本没有机会来做清除工作。读者可以尝试在 `Counter.asm` 例子中使用 `TerminateThread` 函数来终止 `_Counter` 线程的执行。可以发现计数线程是停止了，但是“停止计数”按钮并不会恢复为“计数”按钮，“暂停/恢复”按钮也不会被灰化。因为计数线程平时在循环中执行，被强制终止的时候必然还在循环体内，

这样下面的扫尾代码将没有机会执行，其结果当然如此了：

invoke	SetWindowText, hWinCount, addr szStart
invoke	EnableWindow, hWinPause, FALSE
and	dwOption, not (F_COUNTING or F_STOP or F_PAUSE)

TerminateThread 函数引发的问题可能还有很多，如线程中打开的文件和申请的内存等都不会被释放，更危险的是，如果线程刚好在调用 Kernel32.dll 中的系统函数时被终止，可能会引起 Kernel32 的状态处于不正确的状态（当然只是线程所属进程的 Kernel32 状态而不是系统范围的状态）。另外，当使用 TerminateThread 函数终止线程的时候，系统不会释放线程使用的堆栈。所以建议读者在编程中的时候尽量让线程自己退出，如果主线程要求某个线程结束，可以通过各种方法通知线程，线程收到通知在做扫尾工作后自行退出。只有在迫不得已的情况下，才能使用 TerminateThread 函数去终止一个线程。

第 4 种方法就是使用 ExitProcess 函数结束进程，这时系统会自动结束进程中所有线程的运行。在以前演示的所有的单线程程序中，并不显式地结束主线程的运行，而总是用直接结束进程的方法让主线程自然结束。在多线程的程序中，用这种方法结束线程相当于对每个线程使用 TerminateThread 函数，所以也应当避免这种情况（用这种方法结束主线程的运行并不是问题，因为在这之前可以预测到线程的结束并进行扫尾工作）。

当一个线程终止时，Windows 释放执行线程所需的各种资源，如堆栈与寄存器环境等，并且不再继续分配时间片调用线程中的代码，但线程对象并不马上被释放，因为以后其他线程可能还需要用 GetExitCodeThread 函数检测线程的退出码。线程对象一直保存到使用 CloseHandle 函数关闭线程句柄为止。

4. 其他相关函数

除了上面介绍的一些函数，读者还可以通过其他的相关函数对线程进行控制。下面简单介绍 SuspendThread, ResumeThread 和 GetExitCodeThread 函数的用法。

一个线程可以被挂起（暂停），也可以在挂起后被恢复执行。当使用 CreateThread 函数创建线程的时候，如果在 dwCreationFlags 参数中指定 CREATE_SUSPENDED 标志，线程创建后并不马上开始执行，而是处于被挂起的状态，直到使用 ResumeThread 函数启动它为止。除了在创建的时候直接让线程处于挂起状态，也可以使用 SuspendThread 函数将运行中的线程挂起：

invoke	SuspendThread, hThread
--------	------------------------

该函数的惟一参数是需要挂起的线程句柄。系统为每个线程维护一个暂停计数器，SuspendThread 函数将导致线程的暂停计数增加，当一个线程的暂停计数大于 0 的时候，系统就不会给线程安排时间片，这就相当于将线程挂起，如果函数执行成功，返回值是线程原来的暂停计数值，当函数执行失败时，返回值是 -1。如果创建线程的时候使用 CREATE_SUSPENDED 标志，那么线程的暂停计数值一开始就是 1。

要将挂起的线程恢复到执行状态，可以使用 ResumeThread 函数：

invoke	ResumeThread, hThread
--------	-----------------------

该函数减少线程的暂停计数，当计数值减到 0 的时候，线程被恢复运行，所以函数被调用后线程是否被恢复运行还要看原来的暂停计数值是多少，如果多次调用 `SuspendThread` 函数导致暂停计数值远远大于 1 的话，就必须多次调用 `ResumeThread` 后线程才能被恢复运行。`ResumeThread` 的返回值定义和 `SuspendThread` 函数的定义是一样的。

一个线程可以将别的线程挂起，也可以将自己挂起，但是将自己挂起后，显然不可能再由自己来恢复运行，因为这时线程自己不可能再运行 `ResumeThread` 函数了，在这种情况下，必须由其他线程来进行恢复操作。

在例子程序中，也可以将通过检测标志位来“暂停/恢复”的功能改为使用挂起/恢复计数线程的办法来实现。

`GetExitCodeThread` 函数用来获取线程的退出码，同时也可以用来检测线程是否已经结束。函数的用法是：

invoke	<code>GetExitCodeThread, hThread, lpExitCode</code>
--------	---

其中 `hThread` 参数指定需要获取的线程句柄，`lpExitCode` 指向一个双字变量，用来接收函数返回的退出信息，如果函数执行成功，返回非 0 值，并且将退出码返回到 `lpExitCode` 指向的变量中，如果执行失败，函数返回 0。

当一个线程没有结束的时候，退出信息中返回的是 `STILL_ACTIVE`，如果线程已经结束，那么变量中返回的就是线程的退出码，通过检查退出信息是否为 `STILL_ACTIVE` 就可以得知线程是否已经结束。

12.3 使用事件对象控制线程

12.2.2 节中经过改进的多线程版的 `Counter` 程序运行起来一切正常，但是不知道读者有没有发现一个小缺点——在 CPU 时间占用上的小缺点。

如果程序在 Windows NT 系列操作系统中运行，就可以从任务管理器中发现这个问题（可以通过按下 `Ctrl+Alt+Del` 键调出任务管理器程序），如图 12.2 所示，当计数正在进行的时候，任务管理器显示 `Counter.exe` 程序的 CPU 占用率为 96%，这没有什么奇怪，因为当前只有这一个程序在瞎忙活，并没有其他大运算量的程序，所以 `Counter.exe` 程序占用了绝大部分的 CPU 时间。



图 12.2 Counter 程序的 CPU 占用率

现在按下“暂停/恢复”按钮将计数暂停，就可以看出问题来了——即使计数暂停了，但是程序的 CPU 占用率还是保持不变，根本没有降下来，这是为什么呢？其实不难解释，在 _Counter 子程序中使用下面的语句来检测是否暂停：

```
.if      !(dwOption & F_PAUSE)
        inc     ebx
        invoke  SetDlgItemInt,hWinMain, IDC_COUNTER, ebx, FALSE
.endif
```

当计数暂停的时候，dwOption 的 F_PAUSE 位被设置，这时程序跳过了中间的 inc ebx 指令和 SetDlgItemInt 函数，但是为了随时能够响应用户恢复计数的动作，程序不得不循环检测 dwOption 变量，以至于虽然没有做任何有用功，但还是把所有的 CPU 时间都花在了检测标志上面。

对于这样一个小程序来说，效率不是主要的问题，但如果在一个大型的拥有很多线程的程序中，这就会严重影响效率。对于这种问题，最彻底的解决方法就是让操作系统来决定是否继续执行程序，如果操作系统了解线程什么时候需要等待，什么时候需要执行的话，它就可以仅在线程需要执行的时候安排时间片，在等待的时候干脆连时间片都不用分配，这样就不会在检测标志上浪费时间了。

按照这个思路，使用 SuspendThread 和 ResumeThread 函数来挂起和恢复线程是一个可行的办法，主线程不必通过设置标志位来通知工作线程进入等待状态，而是直接使用 SuspendThread 函数将工作线程挂起就可以了。使用这种方法的好处是可以解决 CPU 利用率的问题，因为操作系统不会给挂起的线程分配时间片，缺点就是无法精确地控制线程，因为主线程不知道工作线程会在哪里被暂停，暂停点可能会在 inc ebx 指令上，也有可能在测试 dwOption 的指令中，甚至在执行 SetDlgItemInt 函数的系统内核中。如果要求工作线程必须在完成整个循环体代码的情况下才能暂停的话，就无法使用这种方法，这时必须在循环体的头部进行条件测试。

难道除了不断地测试暂停标志就没有其他方法了吗？当然不是，下面介绍的事件对象就可以用来解决这个问题。

12.3.1 事件

Windows 中可以创建很多种类的对象，如文件、窗口和内存等对象都是看得见摸得着的实体，事件(Event)也是一种对象，但事件对象比较抽象，我们可以把它看成是一个设置在 Windows 内部的标志，它的状态设置和测试工作由 Windows 来完成，Windows 可以将这些标志的设置和

测试工作和线程调度等工作在内部结合起来，这样效率就要高得多。

事件可以有两种状态：“置位的”和“复位的”。如果想使用事件对象，需要首先使用 `CreateEvent` 函数去创建它，就像在程序中为自己的标志变量分配内存一样：

```
invoke    CreateEvent, lpEventAttributes, bManualReset, bInitialState, lpName
.if      eax
mov      hEvent, eax
.endif
```

函数的参数定义如下：

- `lpEventAttributes` 参数指向一个 `SECURITY_ATTRIBUTES` 结构，用来定义事件对象的安全属性，如果事件对象的句柄不需要被继承，可以在这里指定 `NULL`。
- `bManualReset` 参数指定事件对象是否需要手动复位，如果指定 `TRUE`，对事件对象状态的复位工作必须使用 `ResetEvent` 函数手动完成。指定 `FALSE` 的话，当测试事件的函数返回时（返回原因可能是超时，也可能是对象状态被置位引起），对象的状态会自动被复位。
- `bInitialState` 参数指定事件对象创建时的初始状态，`TRUE` 表示初始状态是置位状态，`FALSE` 表示初始状态是复位状态。
- `lpName` 指向一个以 0 结尾的字符串，用来指定事件对象的名称，和内存共享文件一样，为事件对象命名是为了在其他地方使用 `OpenEvent` 函数获取事件对象的句柄。如果不需要命名，那么可以在这里使用 `NULL`。

如果函数执行成功，函数的返回值是事件的句柄，如果失败，则返回 0。

当一个事件被建立后，程序就可以通过 `SetEvent` 和 `ResetEvent` 函数来设置事件的状态，就像我们使用 `or` 或 `and` 指令将程序中的标志变量置位或复位一样：

```
invoke    SetEvent, hEvent          ;将事件的状态设为“置位”
invoke    ResetEvent, hEvent        ;将事件的状态设为“复位”
```

参数 `hEvent` 就是 `CreateEvent` 函数返回的事件句柄。当不再需要事件对象的时候，可以使用 `CloseHandle` 函数将它释放掉。

12.3.2 等待事件

就像用测试指令来测试标志一样，如果将事件看成是“标志”的话，就需要有函数来实现测试功能，`WaitForSingleObject` 就是这样的函数，注意：函数的名称包含 `Wait`（“等待”）一词而不是“测试”，如果函数仅可以用来测试事件的状态的话，事件对象就失去了使用的初衷，因为这样的话，在线程中循环测试标志的情况又会重演了。

`WaitForSingleObject` 函数的用法是：

```
invoke    WaitForSingleObject, hHandle, dwMilliseconds
```

`WaitForSingleObject` 函数可以测试的不仅是事件对象，它也可以用来测试线程和进程等对象的状态，`hHandle` 参数用来指定为等待的对象句柄，`dwMilliseconds` 参数指定以 `ms` 为单

位的超时时间，当以下两种情况中的任意一种发生的时候，函数就返回：

- 测试对象的状态变为置位状态。
- 到了 dwMilliseconds 指定的超时时间。

如果 dwMilliseconds 参数指定为 0 的话，WaitForSingleObject 在测试对象的状态后马上返回，如果需要函数无限期等待直到对象的状态变为“置位”为止的话，可以在该参数中使用 INFINITE 预定义值。

如果函数执行失败，返回值为 WAIT_FAILED。如果函数执行成功，返回值代表函数返回的原因，当返回值是 WAIT_OBJECT_0 时，表示返回原因是对象的状态被置位，返回值是 WAIT_TIMEOUT 的时候表示返回原因是超时。

函数可以测试的对象有多种，不同的对象对状态的定义是不同的，下面列出了部分函数支持的对象对状态的定义：

- 控制台输入（Console input）——如果用户的输入使控制台的输入缓冲区不为空的时候，控制台对象的状态为“置位”，当输入缓冲区空的时候，状态变为“复位”。
- 事件对象（Event）——对事件对象调用 SetEvent 函数后，状态为“置位”，对事件对象调用 ResetEvent 函数后，状态为“复位”。
- 进程对象（Process）——如果进程结束，状态为“置位”。
- 线程对象（Thread）——如果线程结束，状态为“置位”。

可以看到，WaitForSingleObject 函数也可以很方便地用来等待线程结束，这样当程序必须等待某个线程结束的时候，就不必用一个循环不停调用 GetExitCodeThread 函数，然后通过检测返回值是否还是 STILL_ACTIVE 来判断了。

WaitForSingleObject 函数仅可以测试一个对象，在实际的应用中，还常常会遇到需要同时测试多个对象的情况，这时可以使用另外一个函数：WaitForMultipleObjects。这个函数的用法是：

Invoke WaitForMultipleObjects, dwCount, lpHandles, bWaitAll, dwMilliseconds

lpHandles 指向一组对象句柄变量，对象句柄的数量由 dwCount 参数指定，函数将同时测试这些对象句柄的状态。

bWaitAll 参数用来定义测试的逻辑。如果指定为 TRUE，函数仅在所有对象的状态都变成“置位”时才返回（相当于执行 and 操作）。如果指定为 FALSE，任意一个对象的状态变成“置位”时，函数就会返回（相当于执行 or 操作）。

函数的其他用法，如 dwMilliseconds 参数以及返回值的定义和 WaitForSingleObject 中的定义都是相同的。

12.3.3 进一步改进计数程序

现在让我们进一步改进前面的计数程序，用事件对象代替暂停标志，用

器

- 器

器

器

436

```

        invoke    EndDialog, hWnd, NULL
;*****
        .elseif  eax ==    WM_INITDIALOG
            push    hWnd
            pop     hWinMain
            invoke  GetDlgItem, hWnd, IDOK
            mov     hWinCount, eax
            invoke  GetDlgItem, hWnd, IDC_PAUSE
            mov     hWinPause, eax
            invoke  CreateEvent, NULL, TRUE, FALSE, NULL
            mov     hEvent, eax
;*****
        .else
            mov     eax, FALSE
            ret
        .endif
        mov     eax, TRUE
        ret

_ProcDlgMain    endp
;>>>>>>>>>
start:
        invoke    GetModuleHandle, NULL
        mov       hInstance, eax
        invoke    DialogBoxParam, eax, DLG_MAIN, \
            NULL, offset _ProcDlgMain, NULL
        invoke    ExitProcess, NULL
;>>>>>>>>>
        end       start

```

修改以后编译执行，然后打开任务管理器观察程序的 CPU 占用率就可以发现，当按下“暂停/恢复”按钮暂停计数后，程序的 CPU 占用率马上下降到接近零，再次按动按钮恢复计数时，CPU 占用率会恢复到原来的数值，这说明程序的运行效率得到了很大的提高。

12.4 线程间的同步

12.4.1 产生同步问题的原因

对于多线程程序来说，线程之间的同步永远是个重要的问题。如果多个线程都要存取同样的对象（如存取同样的内存变量或读写同一个文件等），而一个线程操作的结果反过来又会影响另一个线程的运行的时候，同步问题就变得异常重要。

产生同步问题的根源在于线程之间的切换是无法预测的，一个线程无法知道什么时候自己的时间片会结束，也无法知道下一个时间片会被分配给哪个线程。事实上，线程可以在任何地方被 Windows 打断。读者应该记住的是：惟一可以确定的事实就是线程只能在两条指令之间被打断，因为指令是 CPU 执行的最小单位，线程不可能在一条指令执行到一半的时候被打断。

对于单线程的程序来说，主线程在单个时间片结束的时候被 Windows 挂起，然后在轮到下

- (1) 获取账户的余额。
- (2) 将用户存入的数额和余额相加，得到新的余额。
- (3) 将账户中的余额数据更新为新的数值。

有人可能会认为出现这种情况的概率是很低的，线程中有这么多条指令要执行，难道偏偏就在程序取完数据还没开始处理的时候被系统打断吗？通过下面的例子就可以发现发生这种事情的可能性有多大。例子程序位于所附光盘的 Chapter12\ThreadSynErr 目录中，还是用递增计数器的方法来演示同步问题，ThreadSynErr.asm 的内容如下：

哭

440

最后，程序设立一个定时器来定时将计数器值显示到编辑框中。

在线程函数中，使用下面的计算代码：

```
.while    ! (dwOption & F_STOP)
          inc      dwCounter1
          mov      eax, dwCounter2
          inc      eax
          mov      dwCounter2, eax
endw
```

在这段代码中，递增第一个计数器使用 `inc` 指令，由于计算用单条指令完成，所以计数器一不会因为同步问题出错，递增第二个计数器的代码使用了 3 条指令，首先将原来的计数值取到 `eax` 中，递增 `eax` 后再写回到变量中，如果不存在多个线程同步的问题，这两种算法的结果是一样的，显示到编辑框中的计数值应该是相等的。

在存在同步问题的情况下，如果线程在 `mov eax, dwCounter2` 或者 `inc eax` 指令执行后被打断，并且其他线程在这期间修改了 `dwCounter2` 的值的话，根据前面的分析，就会有一次计数值被丢失。如果显示到编辑框中的计数值不相等，则证明存在同步的问题，通过比较两个计数值的差值，还可以得知同步问题发生的机会是多少。

好了，大家可能都迫不及待地想看运行结果了吧，结果如图 12.3 所示，这是程序在笔者的 700 MHz 的计算机上运行了 10 秒以后的结果，可以看到，10 个线程加起来总共进行了 1 174 582 125 次计算，计数器二却丢失了 $1\,174\,582\,125 - 413\,019\,673 = 761\,562\,452$ 次计数，因为同步问题丢失的计数竟然占了 65%，可见这绝对不是偶尔发生一次两次的事情，大家可以想象一下，如果有人往一个银行账户中汇款，三笔汇款中丢了两笔，人们会有何感想呢？

12.4.2 各种用于线程间同步的对象

了解了同步问题产生的根源，再提出解决方案是很简单的，这在其他的应用程序中早有体现，如各种多用户版的数据库在操作记录之前都要对记录进行锁定，保证一条记录在同一时刻只能被一个对象操作；Windows 中的写文件函数在遇到其他程序正在写入的时候会返回共享错误，而不是不管青红皂白直接写入了事。类似的例子还可以找到很多，归纳起来不外乎一点：就是保证整个存取过程的独占性，也就是当一个线程要进行操作前，需要等待其他操作中的线程结束。

在编程上，独占机制不能简单地用一个标志变量来解决，因为测试标志位和改变标志位的过程仍然可能被其他线程打断，导致有多个线程同时认为标志有效并对数据进行操作。如果有种方法，能够保证标志位的测试和改变过程不被打断就行了。

幸亏 Windows 提供了多种同步对象供我们使用，仔细回想本章中介绍的事件对象，就可以发现事件对象符合这个特征，`CreateEvent` 时将 `bManualReset` 参数设置为 `FALSE` 的时候，测试事件的函数返回时，对象的状态会自动被复位，测试和设置状态的过程不会被其他线程打断。除了事件 (Event) 对象外，临界区 (Critical Section)、互斥量 (Mutex) 和信号灯 (Semaphore) 等对象都可以用于线程同步。

哭

哭

哭

哭

哭

444

请读者注意代码中的粗体部分，程序在 WM_INITDIALOG 消息中创建了一个事件对象：

第三个参数是 `TRUE`，这样创建的事件对象的初始状态为置位，保证肯定有一个线程能够通过 `WaitForSingleObject` 函数，如果这个参数是 `FALSE` 的话，事件对象的初始状态是复位，那么所有的线程都会处于等待状态，程序也就无法正常工作了。

将代码经过编译链接后再运行，就可以发现两个计数器的值永远保持统一了！读者还可以发现一个现象：在相同的时间内，正确同步的程序的计数值要远远小于同步不正确的程序，比如在笔者的计算机上，同样是运行 10 秒，这个程序只能计数 2 249 473 次，与前述例子的 1 174 582 125 次相比，前者不及后者的 2%，这说明花在等待上的时间实在是太多了，但是这是多线程程序为了保持数据同步所必须付出的代价。

445

2. 使用临界区对象进行线程间同步

临界区对象 (Critical Section) 是定义在数据段中的一个 CRITICAL_SECTION 结构, 在使用时, 结构的具体字段不必关心, 也不应该关心, 因为它的维护和测试工作都是由 Windows 来完成的, 读者只需把它想像成一个标志就可以了, 结构应当定义成全局变量, 因为在各线程中都要测试它。

定义了 CRITICAL_SECTION 结构后, 必须首先对它进行初始化, 这个步骤类似于使用事件对象时的 CreateEvent 操作:

invoke	InitializeCriticalSection, lpCriticalSection
--------	--

lpCriticalSection 参数指向数据段中定义的 CRITICAL_SECTION 结构。

由于 Windows 操作系统保证临界区对象同时只能被一个线程进入, 所以在需要独占操作的代码前加上进入临界区的操作, 代码后面加上离开临界区的操作, 就可以保证操作的独占性。在上面的例子中使用临界区对象进行同步的话, 可以对计数线程进行如下修改:

invoke	EnterCriticalSection, addr stCS ; stCS 是 CRITICAL_SECTION 结构
inc	dwCounter1
mov	eax, dwCounter2
inc	eax
mov	dwCounter2, eax
invoke	LeaveCriticalSection, addr stCS

进入临界区的操作由 EnterCriticalSection 函数来完成。如果当前有其他线程拥有临界区, 函数不会返回, 反之如果函数返回就表示现在可以独占数据了。调用 EnterCriticalSection 函数可以看成是让 Windows 检测标志, 如果是“不允许”则等待; 是“允许”则将标志修改为“不允许”状态并返回。

当完成操作的时候, 还要将临界区交还 Windows, 以便其他线程可以申请使用, 这个工作由 LeaveCriticalSection 函数完成, LeaveCriticalSection 函数的功能可以看成是将标志从“不允许”改回“允许”状态。

当程序不再使用临界区的时候, 必须使用 DeleteCriticalSection 将它删除:

invoke	DeleteCriticalSection, lpCriticalSection
--------	--

详细的代码也可以在所附光盘的 Chapter12\ThreadSyn\UseCriticalSection 目录中找到。读者可以自行查看其中的细节。

与事件对象不同, 由于临界区对象无法命名, 所以无法跨进程使用, 但正是因为事件对象可以跨进程使用, 需要占用的资源更多, 所以相比之下临界区对象在速度上的优势很明显, 上面的例子中, 使用临界区对象进行同步的程序在 10 秒中之内可以计数 81 659 524 次, 比使用事件对象快 30 多倍。

使用临界区的缺点在于, 如果某个线程进入临界区后挂掉了, 那么将无法被其他等待的线程检测到, 因为这些线程都“陷”在 EnterCriticalSection 函数中了。但是使用事件对象时, 可以在 WaitForSingleObject 函数中指定一个超时时间, 当这个时间设置为一个很宽余的时间

后，函数仍然因为超时而返回，那么就意味着某个线程挂掉了，程序就可以进行相关的处理。

在具体的使用中，使用哪种对象进行线程同步要看具体情况，一般在对速度要求比较高，并且不必跨进程进行同步的情况下，建议使用临界区对象。

3. 使用互斥量对象进行线程间同步

互斥量（Mutex）与临界区的作用相似，也是一种同时只允许一个线程获取的对象，但它是可以命名的，所以可以跨越进程使用。下面首先介绍对互斥量进行操作的函数。

互斥量可以用 CreateMutex 函数创建：

```
invoke    CreateMutex, lpMutexAttributes, bInitialOwner, lpName
.if      eax
    mov    hMutex, eax
.endif
```

函数各参数的定义如下：

- lpMutexAttributes 参数指向 SECURITY_ATTRIBUTES 结构，用来定义互斥对象的安全属性，如果事件对象的句柄不需要被继承，可以在这里指定 NULL。
- bInitialOwner 参数如果设置为 TRUE，表示创建它的线程直接获取了该互斥量，设置为 FALSE 表示互斥量创建后处于空闲状态。
- lpName 指向一个以 0 结尾的字符串，用来指定互斥对象的名称，如果为互斥对象命名，则对象可以在其他进程中用 OpenMutex 函数打开，以便用于进程间的线程同步。如果不需要命名，那么可以在这里使用 NULL。

释放互斥量可以使用 ReleaseMutex 函数，但是调用该函数的线程必须首先拥有该互斥量，这一点和使用 Event 不同，在任何线程中都可以使用 SetEvent 函数来将 Event 对象置位：

```
invoke ReleaseMutex, hMutex
```

不需要再使用的时候，可以使用 CloseHandle 函数来关闭互斥量。

看到这里，读者可能会问，获取互斥量的函数呢？事实上系统中没有专门的类似于 EnterMutex 的函数，互斥量是靠 WaitForSingleObject 函数来获取的。当互斥量被某个线程获取（即等待成功）后，它的状态是复位的，否则是置位的。

在前面的例子中使用互斥量对象进行同步的话，首先用 invoke CreateMutex, NULL, FALSE, NULL 创建一个互斥量，然后对计数线程进行如下修改：

```
invoke    WaitForSingleObject, hMutex, INFINITE
inc       dwCounter1
mov       eax, dwCounter2
inc       eax
mov       dwCounter2, eax
invoke    ReleaseMutex, hMutex
```

WaitForSingleObject 函数等待成功后会将互斥量置为复位，这样其他的线程就无法获取而继续等待，直到线程调用 ReleaseMutex 函数释放互斥量后，对象的状态变为置位，才会有

另一个线程继续得到控制权。

详细的代码可以在所附光盘的 Chapter12\ThreadSyn\UseMutex 目录中找到。读者可以自行查看其中的细节。

使用互斥量进行同步的缺点和使用事件对象类似，也是在效率上远远低于使用临界区；当然优点也是类似的，那就是可以跨进程使用，可以检测到其他线程是否挂掉。

4. 使用信号灯对象进行线程间同步

信号灯（Semaphore）对象是一个允许指定数量的线程获取的对象。信号灯对象一般用于线程排队，考虑这样一种情况：服务器程序设置了 3 个工作线程以及 n 个和客户端连接的服务线程，服务线程需要在工作线程空闲时与之通讯，但 3 个工作线程都忙的时候，就必须等待。这种情况下设置一个计数值为 3 的信号灯对象，服务线程和工作线程通讯前首先尝试获取信号灯，可以保证同时只有 3 个服务线程可以通过，而其他线程处于等待状态。

当信号灯对象的计数值设置为 1 的时候，就相当于一个互斥量。

首先来看看和信号灯对象相关的函数，信号灯对象可以用 CreateSemaphore 函数创建：

```

invoke    CreateSemaphore, lpSemaphoreAttributes, dwInitialCount, \
          dwMaximumCount, lpName
. if      eax
mov       hSemaphore, eax
. endif

```

函数各参数的定义如下：

- lpSemaphoreAttributes 参数指向 SECURITY_ATTRIBUTES 结构，用来定义信号灯对象的安全属性，如果事件对象的句柄不需要被继承，可以在这里指定 NULL。
- dwInitialCount 参数表示初始化的时候对象的计数值是多少，也就是说还有多少个线程允许获取对象，这个值必须大于等于 0 并且小于等于下面的 dwMaximumCount 参数。
- dwMaximumCount 参数表示对象的最大计数值是多少，也就是最多允许多少个线程获取该对象。
- lpName 指向一个以 0 结尾的字符串，用来指定对象的名称，如果为信号灯对象命名，则对象可以在其他进程中用 OpenSemaphore 函数打开，以便用于进程间的线程同步。如果不需要命名，那么可以在这里使用 NULL。

释放信号灯对象使用 ReleaseSemaphore 函数，其中的 dwReleaseCount 指定释放的时候将计数值增加多少（一般使用 1），lpPreviousCount 指向一个双字变量，函数在此返回释放前的计数值，如果不需要返回这个值，可以将此参数设置为 NULL：

```

invoke ReleaseSemaphore, hSemaphore, dwReleaseCount, lpPreviousCount

```

不需要再使用的时候，可以使用 CloseHandle 函数来关闭信号灯对象。

同样，信号灯对象是靠 WaitForSingleObject 函数来获取的。当对象被某个线程获取后，

它的计数值减 1，当计数值未减到 0 的时候，对象的状态是置位的，这意味着还有线程可以继续获取该对象；当计数值减到 0 的时候，对象的状态被复位。

在前面的例子中，如果使用信号灯对象进行同步，可以首先用 `invoke CreateSemaphore, NULL, 1, 1, NULL` 创建对象，对象的最大计数值和初始计数值都设置为 1，然后对计数线程进行如下修改：

<code>invoke</code>	<code>WaitForSingleObject, hSemaphore, INFINITE</code>
<code>inc</code>	<code>dwCounter1</code>
<code>mov</code>	<code>eax, dwCounter2</code>
<code>inc</code>	<code>eax</code>
<code>mov</code>	<code>dwCounter2, eax</code>
<code>invoke</code>	<code>ReleaseSemaphore, hSemaphore, 1, NULL</code>

`WaitForSingleObject` 函数等待成功后会将信号灯对象的计数值减 1，由于初始计数值是 1，所以一旦有一个线程获取对象后，对象的状态即变为复位，其他的线程就无法获取而继续等待，直到线程调用 `ReleaseSemaphore` 函数将计数值增 1 为止，对象的状态变为置位，才会有另一个线程继续得到控制权。

详细的代码可以在所附光盘的 `Chapter12\ThreadSyn\UseSemaphore` 目录中找到。读者可以自行查看其中的细节。

第 13 章

进程的概念在第 12 章中有所介绍，所谓进程，就是一个执行中的文件所使用的资源的总和，包括虚拟地址空间、代码、数据、对象句柄、环境变量和用来执行代码的线程等。只产生一个新的进程是很简单的，说穿了就是在一个程序中执行另一个程序而已，在 DOS 操作系统下编程的时候我们就对执行另一个文件使用的 `int 21h/4bh` 功能非常熟悉。

如果只为了介绍如何执行一个文件，那么就根本不需要使用单独的一章，实际上，有关进程的课题中最令人感兴趣的是如何在建立了一个进程后继续控制它，包括对它进行跟踪调试，读写其他进程的地址空间及进程的隐藏等，本章的重点就是介绍这些内容。

13.1 节首先介绍一些与进程相关的周边知识，包括如何使用命令行参数和如何操作环境变量等。13.2 节介绍进程的创建、终止和等待。13.3 节介绍与进程调试有关的内容，包括如何枚举系统中运行的进程，读写进程的地址空间和调试 API 的使用等。13.4 节的内容与病毒、木马等有害程序的防治有关，中间演示了这些有害程序常用的进程隐藏与创建远程线程等功能的实现，使读者对这些内容有了初步的了解。

13.1 环境变量和命令行参数

13.1.1 环境变量

1. 什么是环境变量

环境变量就是在命令提示符下键入“Set”命令后列出来的内容，它的定义格式以 `XXX=YYY` 的形式来表示，其中的 XXX 是环境变量的名称，YYY 是环境变量的值，下面的例子是笔者使用的计算机中列出的部分环境变量：

```
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Application Data
BLASTER=A220 I7 D1 H7 P330 T6
SBPCI=C:\SBPCI
COMPUTERNAME=WORKGROU-86NSVP
```

```

ComSpec=C:\WINDOWS\system32\cmd.exe
SystemDrive=C:
SystemRoot=C:\WINDOWS
HOMEDRIVE=C:
ProgramFiles=C:\Program Files
HOMEPATH=\Documents and Settings\Administrator
LOGONSERVER=\\WORKGROU-86NSVP
OS=Windows_NT
Path=C:\WINDOWS\system32;C:\WINDOWS;c\tools;C:\WIN98\Twain_32\Nuscan
NUMBER_OF_PROCESSORS=1
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 6 Stepping 0, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0600
PROMPT=$P$G
TEMP=C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
TMP=C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp
USERDOMAIN=WORKGROU-86NSVP
USERNAME=Administrator
windir=C:\WINDOWS

```

环境变量按照用途可以分为 3 大类：

- 与系统运行相关的环境变量——这些变量的值与系统的正常运行息息相关。如 PATH 变量定义的是可执行文件的搜索路径，它直接影响系统搜寻可执行文件的位置和先后顺序；而 TEMP 和 TMP 变量将影响系统创建临时文件的位置；ComSpec 变量定义命令行管理器的文件名，在 DOS 操作系统下，如果这个变量的定义错误，就会导致系统无法装入 Command.com 而挂起。
- 反映系统状态的环境变量——如 NUMBER_OF_PROCESSORS，PROCESSOR_LEVEL 和 PROCESSOR_IDENTIFIER 等是操作系统根据当前的硬件定义的；COMPUTERNAME，USERNAME 和 USERDOMAIN 等变量是操作系统根据当前的机器名、登录用户等定义的；OS，HOMEDRIVE 和 ProgramFiles 等则定义了操作系统的名称和安装位置。通过检测这些变量，应用程序可以了解系统的配置情况和其他一些重要信息。
- 应用程序自定义的环境变量——如上面列出的 BLASTER 和 SBPCI 是创新声卡自定义的变量。在本书例子程序的编译环境中，也定义了一些编译器和链接器使用的变量，如 Include 与 Lib 等，这些变量是应用程序根据自己的需要自定义的。

2. 对环境变量进行操作

在命令提示符窗口中，可以通过“Set 变量=内容”格式的命令来设置环境变量的值，也可以通过等号后面不带内容的“Set 变量=”命令来删除某个环境变量。如果需要在程序中对环境变量进行操作，可以使用 Win32 中的以下几个 API。

如果想获取一个环境变量的值，可以使用 GetEnvironmentVariable 函数：

```
invoke GetEnvironmentVariable, lpVarName, lpBuffer, dwSize
```

lpVarName 指向一个以 0 结尾的字符串，用来指定需要获取的环境变量名，lpBuffer 指向用来接收变量值的缓冲区，dwSize 参数指定缓冲区的大小。如果函数执行成功，返回值是返回到缓冲区中的字符数量（不包括结束的 0 字符）；如果环境变量不存在，返回值是 0；如果缓冲区太小以至于放不下环境变量内容，那么缓存区中不会返回任何内容，这时函数的返回值是需要的缓存区的大小，这就意味着，如果返回值比缓冲区的大小要大，那么必须扩大缓冲区后再次调用。

下面的代码演示了如何将 PATH 变量的值获取到 szBuffer 缓冲区中：

		.data
szBuffer		db 200 dup (?)
szVarName	db	'PATH', 0
		.code
	invoke	GetEnvironmentVariable, addr szVarName, \
		addr szBuffer, sizeof szBuffer

GetEnvironmentVariable 函数只能用来获取已知名称的环境变量，如果需要枚举所有的环境变量，可以使用 GetEnvironmentStrings 函数。这个函数返回一个内存块指针，内存块中包含了所有的环境变量定义，通过扫描整个内存块就可以获得所有的环境变量定义。该函数没有输入参数：

invoke	GetEnvironmentStrings
mov	lpVar, eax

内存块中环境变量存放格式为“变量 1=内容 1”，0，“变量 2=内容 2”，0，……“变量 N=内容 N”，0，0，即每个定义字符串以 0 结束，然后开始下一个变量定义字符串，全部定义字符串的最后再以一个附加的 0 结束。

GetEnvironmentStrings 函数返回的内存块是系统申请的，当不需要再使用的时候，需要将它释放，释放这个内存块并不等于删除全部环境变量，而仅是释放这份拷贝而已。释放使用的函数是 FreeEnvironmentStrings：

invoke	FreeEnvironmentStrings, lpVar
--------	-------------------------------

函数的输入参数 lpVar 就是 GetEnvironmentStrings 函数返回的内存块指针。

如果需要改变现存环境变量的值，设置新的环境变量或者删除某个环境变量，可以使用 SetEnvironmentVariable 函数：

invoke	SetEnvironmentVariable, lpVarName, lpValue
--------	--

lpVarName 指向环境变量的名称，lpValue 指向一个以 0 结尾的字符串，用来指定环境变量的新值。

当 lpVarName 指定的环境变量已经存在且 lpValue 指向一个空串时，这个变量将被删除；如果 lpValue 指向的不是一个空串，那么环境变量的值将被改为这个新的字符串；如果 lpVarName 指定的环境变量不存在且 lpValue 指向的不是一个空串，那么系统将建立新的环境变量。

SetEnvironmentVariable 函数的运行结果仅改变本进程的环境变量，并不会影响其他进程。比如，打开一个命令提示符窗口，在其中改变某些环境变量的设置，然后再打开另一个命令提示符窗口查看，就会发现这个新窗口中的环境变量并没有改变。

但是环境变量的值可以被子进程继承，如果在一个程序中创建了另一个进程，那么可以让这个子进程“看到”改动以后的环境变量，这就是在一个命令提示符窗口中改变了环境变量的设置，然后用命令行方式运行一些程序，改变的环境变量对这些程序的运行都有效的原因。

13.1.2 命令行参数

1. 什么是命令行参数

在命令行中通过输入文件名来执行文件，在文件名后面跟的参数就是命令行参数，比如，通过 Telnet 连接到某个远程计算机的 1234 号端口，可以输入：

```
Telnet 192.168.0.1 1234
```

“Telnet”后面跟的“192.168.0.1 1234”就是命令行参数，它可以被程序获取，命令行参数是当做一个以 0 结尾的字符串被程序获取的。

对于窗口程序来说，命令行参数并不是必需的，因为大部分的窗口程序都在菜单中提供了“打开文件”、“选项”等功能，并不需要用户在命令行参数中指定，试想一下：如果窗口程序必须依靠命令行参数输入某些数据的话，那么用户就不能通过在任务管理器中双击图标来执行程序了，因为这样无法输入命令行参数。

但是某些情况下，命令行参数又是窗口程序的必然补充，比如，Windows 中的文本文件往往被关联到记事本程序 Notepad.exe 上，直接双击文本文件，Windows 就会自动执行 Notepad.exe，并把文本文件名通过命令行参数传递给它，所以对于 Notepad.exe 来说，虽然已经在菜单中提供了“打开文件”功能，但也必须处理命令行参数，否则对关联文件就无法处理。

而对于控制台程序（如系统中的 Ping.exe，Format.exe 和 Xcopy.exe 等在命令行下运行的程序）来说，它们没有自己的窗口，也就无法通过菜单来选择某些功能，这时通过命令行传递各种参数就是必然的选择。

2. 使用命令行参数

要获取命令行参数，可以使用 GetCommandLine 函数，这个函数没有输入参数，返回值是一个指向命令行参数字符串的指针：

```
invoke  GetCommandLine
mov     lpCmdline, eax
```

获取命令行参数字符串以后，首先必须对它进行处理，比如，对于上面的 Telnet 程序来说，第一个参数“192.168.0.1”指定主机名，第二个参数“1234”指定端口号，但是我们得到的却是一个连在一起的字符串，所以必须扫描字符串将两个参数分开后才能使用，而且，必须通过检查参数字符串防止用户输入了错误的参数。

Win32 中并没有通用的用来扫描参数字符串的函数,有一个函数 `CommandLineToArgvW` 虽然可以用来对字符串进行扫描,但这个函数仅适用于 Unicode 字符串,而且仅在 NT 系列中得到支持,在 Windows 9x 系列中无法使用,为了使用这个函数而将程序限制在 Windows NT 下运行显然是得不偿失的。

C 语言为用户考虑到了这一点,在 C 的初始化程序将控制权交到 `WinMain` 函数之前就已经对命令行参数进行了处理,并将参数按照空格划分成不同的部分放到 `argv` 数组中,参数数量则存放在 `argc` 变量中,在程序的任何地方都可以存取这些变量;但在 Win32 汇编中,这些工作就需要自己来做了,笔者在本节的例子程序中提供了两个通用函数,读者可以把它们用在其他程序中。在分析这个例子之前,先来看看在命令行参数字符串中究竟可以收到哪些内容,这些内容将决定分析字符串的算法。

读者可以编写一个很简单的测试程序 `Test.exe`,在程序中调用 `GetCommandLine` 函数并将得到的命令行参数显示出来,假如把这个 `Test.exe` 放在“C:\Program files”目录下并使用不同的方法去执行它,执行的时候带 3 个参数“aaa bbb ccc”,就可以发现得到的命令行参数可能是下面的样子:

```
(1) test aaa bbb ccc
(2) test.exe aaa bbb ccc
(3) te"st".exe aaa bbb ccc
(4) c:\program " "files\test aaa bbb ccc
(5) "C:\Program files\Test.exe"
(6) "C:\Program files\Test.exe" "C:\aaa bbb ccc"
```

结果(1)到(4)显示的命令行参数字符串是在命令提示符下输入同样的执行命令得到的。需要注意的有两点:

首先是可执行文件名被当做命令行参数的第一个组成部分被传递过来了,所以要使用真正由用户输入的命令行参数,必须首先将这部分过滤掉。

其次是对可执行文件名的处理,输入 `test` 或 `test.exe` 可以正常执行文件并不奇怪,奇怪的是输入 `te " st ".exe` 或者 `c:\program " " files\test` 这样的文件名也能执行,难道这也是合法的文件名吗?

答案是肯定的,这是因为 Windows 将空格当做参数的分隔符,但是长文件名中同样可以使用空格,这就产生了冲突,为了解决这个问题,Windows 使用双引号来界定,假如文件名中存在空格的话,必须将空格包含在两个双引号中,但双引号并不是只能用一对,也并不一定要用在文件名的头尾,实际上它们可以在文件名的任何地方出现,只要是成对的并且所有空格都被包含在某一对双引号中就可以了,双引号本身不是文件名的组成部分,Windows 在最后会自动将它们剔除,但是传递到命令行参数中的时候还是保留了输入时的样子。

结果(5)是在文件管理器中通过双击 `Test.exe` 执行时得到的命令行字符串,结果(6)是在 C 盘的根目录下建立了一个名为“aaa bbb ccc”的文件,并在“打开方式”中选择 `Test.exe` 后得到的命令行字符串。从这些结果中可以发现:只要由 Windows 来打开文件,那么 Windows

哭

根据这些特征，就不难写出一个通用的分析命令行参数的子程序来：

[illegible]

[illegible]

这两个通用子程序被存放在_CmdLine.asm 文件中, 读者可以在其他的程序中用 include 语句将它包含使用, 其中函数_argc 返回命令行参数的个数, 当执行文件时没有附带参数的时候, 函数的返回值一般是 1, 这时获取的命令行字符串中仅有一个组成部分——那就是可执行文件名; argv 函数则将指定编号的参数返回到一个缓冲区中, 读者可以这样使用:

其中 dwArgv 参数指定要获取的参数编号, 0 表示获取字符串中的第一个组成部分 (一般是文件名), 1 表示获取第二个组成部分, 也就是在文件名后面输入的第 1 个参数, 以此类推, 函数对返回的字符串已经做了处理, 丢弃了中间或者两端的所有双引号。lpReturn 指向用来接收参数字符串的缓冲区, dwSize 指定了缓冲区的大小。

[illegible]

[illegible]

程序很简单，首先调用 `GetModuleFileName` 函数获取可执行文件的文件名，这是为了方便读者和参数中获取的文件名做个对比，然后程序调用 `_argc` 函数获得参数数量，并根据这个数量循环获取每个参数。编译链接后输入命令：

```
cmdline aaa bbb "ccc ddd" eee
```



程序会显示出如图 13.1 所示的消息框。

可见，函数正确划分了命令行参数字符串中的各个参数。需要说明的是，为了能够在某个参数中使用空格，函数同样规定可以将参数中的空格用双引号包含，所以参数字符串中的 " ccc ddd " 被解释为一个参数并丢弃了两端的双引号。

图 13.1 命令行参数例子运行结果

13.2 执行可执行文件

13.2.1 方法一：Shell 调用

Win32 中可以通过 ShellExecute 和 WinExec 函数来执行另一个可执行文件,本节介绍这两个函数的用法。WinExec 函数的使用方法是:

invoke	WinExec, lpCmdLine, dwCmdShow
--------	-------------------------------

lpCmdLine 参数指向一个以 0 结尾的字符串，这个字符串中包含可执行文件加上命令行参数，如果被执行的文件会显示一个窗口，那么函数可以在 dwCmdShow 参数中指定窗口的显示方式，这个参数的定义同 ShowWindow 函数中的 dwCmdShow 参数。

如果文件被成功执行，那么函数返回一个大于 31 的值。使用 WinExec 函数执行文件和在 Windows “开始” 菜单的 “运行” 中键入命令在效果上是一样的。

另一个函数 ShellExecute 的功能相对比较多一点，函数的语法是：

invoke	ShellExecute, hWnd, lpOperation, lpFile, lpParam, \
	lpDirectory, dwCmdShow

这个函数既可以用来执行一个可执行文件，也可以指定一个数据文件名让 Windows 自动查找关联到这个数据文件的可执行文件，并执行这个可执行文件来处理指定的数据文件，数据文件名会以命令行参数的方式传递给可执行文件。函数的参数定义如下。

- hWnd——用来指定被执行文件显示的窗口所属的父窗口。
- lpFile——用来指定文件名，文件名既可以是可执行文件也可以是数据文件。
- lpOperation——指向一个表示执行方式的字符串，字符串可以是下列取值：
 - “open”——文件被打开，这时 lpFile 指定的文件名可以是可执行文件、目录名或数据文件名。如果 lpOperation 参数为空，函数默认执行 open 操作。
 - “print”——文件被打印，这时 lpFile 指定的文件名必须是数据文件。如果指定的是可执行文件，那么函数当做 “open” 操作。
 - “explore”——浏览 lpFile 参数中指定的目录。
- lpParameters——当 lpFile 参数指定了一个可执行文件，本参数用来指定命令行参数。如果 lpFile 参数指定的是数据文件，那么本参数必须是 NULL。
- lpDirectory——执行或打开文件时使用的默认目录。
- dwCmdShow——如果函数执行了一个可执行文件，这个参数指定窗口的打开方式。

如果文件被成功执行，那么函数返回一个大于 31 的值。这里是几个使用 ShellExecute 函数的例子：

.data			
szFileName	db	'Test.exe', 0	
szCmdline	db	'aaa bbb ccc', 0	
szEmail	db	'mailto:luoyunbin@hotmail.com', 0	
szWebPage	db	'http://asm.yeah.net', 0	
szHelp	db	'Win32asm.chm', 0	
.code			
invoke	ShellExecute, 0, 0, addr szFileName, addr szCmdline, 0, SW_SHOWNORMAL		(1)
invoke	ShellExecute, 0, 0, addr szEmail, 0, 0, SW_SHOW		(2)
invoke	ShellExecute, 0, 0, addr szWebPage, 0, 0, SW_SHOW		(3)
invoke	ShellExecute, 0, 0, addr szHelp, 0, 0, SW_SHOW		(4)

例子（1）执行 Test.exe 文件，并将命令行参数“aaa bbb ccc”传递给它。例子（2）打开默认的邮件收发程序并显示一个“新建邮件”窗口，指定的邮件地址会被自动地填入到收件人一栏中。例子（3）打开浏览器并自动打开网站 <http://asm.yeah.net>。例子（4）会运行 chm 帮助文件阅读器 hh.exe，并自动打开 Win32asm.chm 帮助文件。

13.2.2 方法二：创建进程

用 ShellExecute 和 WinExec 函数来执行文件是很方便的，调用这两个函数从某种意义上讲，相当于创建了新的进程，但是函数返回以后，这些新建的进程却脱离了我们的控制，我们无法知道它们在什么时候结束，也无法去强制结束它们。要对进程进行后续的控制，必须使用函数 CreateProcess 来创建进程。

当一个进程被创建的时候，系统进行以下的操作：

- 系统为进程创建一个内核对象，并将它的初始计数设置为 1，与线程对象类似，进程对象只是一个比较小的数据结构，它包含进程的一些统计信息。进程对象可以通过进程句柄来引用。
- 系统为进程创建一个虚拟地址空间，并将可执行文件装载到这个地址空间中。系统同时处理可执行文件的导入表，将导入表中登记的所有 dll 文件装入。每个 dll 文件被装入的时候，DLL 的入口函数被执行，如果入口函数返回初始化失败信息的话，进程的初始化失败。可执行文件本身和所有的 dll 文件都被看做是单独的模块，都被分配了一个实例句柄（实例句柄在数值上等于模块装入到地址空间中的线性地址）。
- 系统为进程建立一个主线程，主线程将从可执行文件的入口地址开始执行。

对于线程来说，Windows 为系统中的每个线程分配一个线程句柄和线程 ID 以便区分它们，同样，对于进程来说，每个进程也对应一个进程句柄和一个进程 ID。

当某个进程创建了一个新的进程的时候，被创建的进程称为“子进程”，创建它的进程称为“父进程”，子进程可以从父进程那里继承环境变量以及其他一些对象，在子进程中可以继续创建“孙进程”。

创建进程使用 `CreateProcess` 函数。下面是一个使用这个函数的例子程序，程序显示如图 13.2 所示的对话框，允许用户输入需要执行的文件名（或者通过“浏览”按钮选择文件名）和传递给文件的命令行参数，当文件开始执行时，程序将“浏览”按钮与文件名输入框等子窗口



图 13.2 建立进程的例子程序

灰化,在子进程结束以后再恢复它们。程序也可以通过“终止”按钮强制结束子进程的运行(“执行”按钮在子进程开始执行后被改为“终止”按钮)。

程序的源代码在所附光盘的 Chapter13\Process 目录中, 其中的 Process.rc 文件定义了上面所示的对话框:

[illegible]

[illegible]

stStartUpSTARTUPINFO <?>

[illegible]

```

;*****
: 恢复按钮状态

```

* ****

```

invoke    RtlZeroMemory addr stProcInfo sizeof stProcInfo

```

```

invoke    RtlZeroMemory, addr stProcInfo, sizeof stProcInfo
invoke    GetDlgItem, hWndMain, IDC_FILE
invoke    EnableWindow, eax, TRUE
invoke    GetDlgItem, hWndMain, IDC_CMDLINE
invoke    EnableWindow, eax, TRUE
invoke    GetDlgItem, hWndMain, IDC_BROWSE
invoke    EnableWindow, eax, TRUE
invoke    SetDlgItemText, hWndMain, IDOK, addr szStart
ret

```

ProcExec endp

[illegible]

```

_ProcDlgMain      proc      uses ebx edi esi hWnd,wMsg,wParam,lParam
                  local     @dwThreadId
                  local     @stOF:OPENFILENAME

```

```
mov     eax, wMsg
```

```
.if     eax == WM_COMMAND
```

```
mov     eax, wParam
```

:*****

```

      .if      ax ==      IDOK

```

```
.if      stProcInfo.hProcess
```

```
invoke TerminateProcess, stProcInfo.hProcess, -1
```

```

else

```

```

invoke    CreateThread, NULL, 0, offset _ProcExec, NULL, \
          NULL, addr @dwThreadID
invoke    CloseHandle, eax

```

, endif

```

    .elseif ax == IDC_BROWSE

```

；浏览打开的文件

```
invoke    RtlZeroMemory, addr @st0F, sizeof @st0F
```

```
mov    @stOF.lStructSize, sizeof @stOF
```

```
push    hWinMain
```

```
pop      @st0F.hwndOwner
```

```
mov     @st0F.lpstrFilter,offset szFileExt
```

```
mov     @st0F.lpstrFile, offset szFileName
```

```
mov     @stOF.nMaxFile, MAX_PATH
```

```
mov     @stOf.Flags, OFN_FILEMUSTEXIST or OFN_PATHMUSTEXIST
```

```
invoke    GetOpenFileName, addr @stOF
```

```
.if      eax
```

```
invoke    SetDlgItemText, hWnd, IDC_FILE, addr szFileName
```

```
.endif
```

;*****

```
.elseif ax == IDC_FILE
```

invoke GetWindowTextLength, lParam

```
mov     ebx, eax
```

invoke GetDlgItem, hWnd, IDOK

[illegible]

当按下“浏览”按钮（IDC_BROWSE）的时候，程序在 WM_COMMAND 消息中显示一个“打开文件”通用对话框并让用户选择可执行文件。

当按下 IDOK 按钮的时候，如果有进程在执行中（hProcess 不为 0），表示现在按下的是“终止”按钮，这时程序调用 TerminateProcess 函数强制结束进程；如果没有进程在执行，表示按下的是“执行”按钮，程序创建一个新线程_ProcExec 子程序，在这个子程序中完成创建新进程和等待它结束的工作。

函数 `CreateProcess` 在子进程创建以后是马上返回的，但是程序需要等待子进程结束，为了在等待的过程中主线程还能够处理对话框消息，程序在这里使用一个新的线程来完成创建和等待子进程的工作。

在线程函数_ProcExec 中，程序灰化“文件名”输入框、“命令行”输入框和“浏览”按钮，并将文件名和命令行参数获取到缓冲区 szFileName 和 szCmdLine 中，接下来调用 CreateProcess 函数创建进程。

1. 创建进程

创建进程需要为新进程窗口的外观指定一些属性，就像使用 Shell 调用方式执行文件时的 dwCmdShow 参数一样，这些属性通过一个 STARTUPINFO 结构来指定：

STARTUPINFO STRUCT				
cb	DWORD	?		;结构的长度
lpReserved	DWORD	?		;保留字段
lpDesktop	DWORD	?		;NT 下使用，指定桌面名称
lpTitle	DWORD	?		;控制台程序使用，指定控制台窗口标题
dwX	DWORD	?		;当新进程使用 CW_USEDEFAULT 参数创建
dwY	DWORD	?		;窗口的时候将使用这些位置和大小属性
dwXSize	DWORD	?		
dwYSize	DWORD	?		
dwXCountChars	DWORD	?		;控制台程序使用，指定控制台窗口行数
dwYCountChars	DWORD	?		
dwFillAttribute	DWORD	?		;控制台程序使用，指定控制台窗口背景色
dwFlags	DWORD	?		;标志
wShowWindow	WORD	?		;窗口的显示方式
cbReserved2	WORD	?		
lpReserved2	DWORD	?		
hStdInput	DWORD	?		;控制台程序使用：几个标准句柄
hStdOutput	DWORD	?		
hStdError	DWORD	?		
STARTUPINFO ENDS				

在需要定制新进程的窗口的时候，才需要手工填写 STARTUPINFO 结构（比如，需要将控制台程序的输入和输出重新定位时，可以改写 hStdInput 和 hStdOutput 字段），在大部分情况下，并不需要新进程的窗口有什么特殊之处，这时只要使用 GetStartupInfo 获取当前进程的 STARTUPINFO 并使用它的默认值就可以了：

.data?			
stStartUp	STARTUPINFO	<?>	
.code			
	invoke	GetStartupInfo, addr stStartUp	

获取 STARTUPINFO 结构以后，就可以把它用在创建进程的函数 CreateProcess 中：

invoke	CreateProcess, lpApplicationName, lpCommandLine, \
	lpProcessAttributes, lpThreadAttributes, bInheritHandles, \
	dwCreationFlags, lpEnvironment, lpCurrentDirectory, \
	lpStartupInfo, lpProcessInformation

函数的各个参数定义如下。

- lpApplicationName——指向一个以 0 结尾的字符串，用来指定可执行文件名，如果这个参数指定为 NULL，那么文件名可以在 lpCommandLine 参数指定的命令行参数中包括。
- lpCommandLine——指向一个以 0 结尾的字符串，用来指定命令行参数，如果 lpApplicationName 参数为 NULL，那么命令行字符串的第一个组成部分用来指定可执行文件名；如果两个参数都不为空，那么 lpApplicationName 用做文件名，lpCommandLine 用做命令行参数。
- lpProcessAttributes——指向一个 SECURITY_ATTRIBUTES 结构，用来指定新进程

的安全属性，如果进程句柄不需要被其他子进程继承，可以在这里使用 NULL。

- `lpThreadAttributes`——指向一个 `SECURITY_ATTRIBUTES` 结构，用来指定新线程的安全属性，如果进程句柄不需要被其他线程继承，可以在这里使用 NULL。
- `bInheritHandles`——指定当前进程的句柄是否可以被新进程继承，如果指定 TRUE，那么可以继承，一般在这里使用 FALSE。
- `dwCreationFlags`——创建标志，指定新进程的优先级以及其他标志，这个参数类似于 `CreateThread` 函数中的同名参数，它可以是一些标志的组合，下面列出了一些常用的标志：
 - `CREATE_NEW_CONSOLE`——如果新进程是控制台程序，那么为它新建一个控制台窗口，而不是使用父进程的控制台窗口。
 - `CREATE_SUSPENDED`——新建进程的主线程一开始处于挂起状态，需要以后用 `ResumeThread` 函数来恢复它的执行。
 - `DEBUG_PROCESS` 和 `DEBUG_ONLY_THIS_PROCESS`——调试进程，相关内容在 13.3 一节的进程调试中会有详细介绍，如果同时指定 `DEBUG_ONLY_THIS_PROCESS` 标志，那么被调试的进程仅是被创建的子进程，否则子进程创建的“孙进程”也在被调试之列。
 - `HIGH_PRIORITY_CLASS`，`IDLE_PRIORITY_CLASS`，`NORMAL_PRIORITY_CLASS` 和 `REALTIME_PRIORITY_CLASS`——用来指定新进程的优先级。
- `lpEnvironment`——指向新进程的环境变量块，如果这个参数指定为 NULL，表示让 Windows 拷贝当前进程的环境块当做子进程的环境块，如果程序需要将修改过的环境块传递给子进程，可以设置这个参数。
- `lpCurrentDirectory`——指向一个路径字符串，用来指定子进程的当前驱动器和当前目录，如果指定为 NULL，子进程将引用父进程的当前路径。
- `lpStartupInfo`——指向前面介绍的 `STARTUPINFO` 结构。
- `lpProcessInformation`——指向一个 `PROCESS_INFORMATION` 结构，这个结构用来供函数返回新建进程的相关信息。

如果函数执行成功，返回值是非 0 值，否则函数返回 0。新建进程的句柄在哪里呢？这些句柄就在 `lpProcessInformation` 参数指向的 `PROCESS_INFORMATION` 结构中。结构定义为：

```
PROCESS_INFORMATION STRUCT
    hProcess      DWORD    ?      ;进程句柄
    hThread       DWORD    ?      ;进程的主线程句柄
    dwProcessId   DWORD    ?      ;进程 ID
    dwThreadId    DWORD    ?      ;进程的主线程 ID
PROCESS_INFORMATION ENDS
```

因为新进程被创建的时候其主线程也同时被创建，主线程句柄也常常会被用到，所以函数要返回的值不仅仅是进程句柄，因此，单靠函数的返回值是无法返回足够的信息的，这就是

CreateProcess 函数用 PROCESS_INFORMATION 结构来返回信息的原因。同样，可以通过检测 PROCESS_INFORMATION 结构是否被填写来判断函数是否执行成功。

理解了这些参数的含义，就会发现 CreateProcess 函数的使用其实是很简单的，因为大部分参数都可以用默认值。例子程序中用下面的代码来创建新进程，读者可以看到大部分的参数都使用默认的 NULL：

```
invoke    lstrcpy, addr @szBuffer, addr szFileName
.if      szCmdLine
    invoke    lstrcat, addr @szBuffer, addr szBlank
    invoke    lstrcat, addr @szBuffer, addr szCmdLine
.endif
invoke    GetStartupInfo, addr stStartup
invoke    CreateProcess, NULL, addr @szBuffer, NULL, NULL, NULL, \
        NORMAL_PRIORITY_CLASS, NULL, NULL, addr stStartup, addr stProcInfo
```

例子代码中将 lpApplicationName 参数设置为 NULL，并将文件名 szfileName 和命令行参数 szCmdLine 合成一个字符串存放在@szBuffer 中，然后一并在 lpCommandLine 参数中指定，为什么不直接使用两个字符串呢，就像下面的代码一样：

```
invoke    CreateProcess, addr szFileName, addr szCmdLine, NULL, NULL, NULL, \
        NORMAL_PRIORITY_CLASS, NULL, NULL, addr stStartup, addr stProcInfo
```

这是因为，这种用法在某种情况下可能引起错误。来做下面的实验：

首先将例子程序中的 CreateProcess 改成分开使用文件名和参数字符串，然后用这个程序去执行 13.1.2 一节中的 Cmdline.exe 程序，并尝试输入不同的内容就可以发现，当不指定命令行参数的时候，运行结果如图 13.3 左图所示：Windows 会自动将文件名当做命令行参数的第一个组成部分传递给子进程，一切正常。

但是指定了命令行以后，问题就出来了，右图是输入参数“aaa bbb ccc”时的结果，也就是说，当指定了命令行以后，Windows 就不会自动在前面加上文件名了，假如被执行的文件将命令行中的第一项当做文件名来看待并将它忽略的话，就会丢失一个参数，遗憾的是，几乎所有的程序都是这样做的！读者可以通过这种方法试将一个文本文件名传递给 Windows 自带的 Notepad.exe，结果就是 Notepad.exe 把它给丢弃了。

为了避免这个错误，程序需要将文件名添加到命令行字符串的前面，但这样的话，指定 lpApplicationName 参数也就变得多此一举了，因为这时不用指定这个参数函数也可以正常运行。（不排除新版本的 Windows 会修正这个错误的可能，但是为了避免在当前版本的 Windows 中出现错误，建议读者用本书例子中的方法来创建进程）

2. 结束进程



图 13.3 CreateProcess 中的命令行

要结束一个进程的执行，可以使用 `ExitProcess` 函数。对于我们来说，这个函数是最熟悉的，因为在所有的例子程序中都用它来结束程序的执行：

invoke	ExitProcess, dwExitCode
--------	-------------------------

与线程结束时有个退出码类似，进程结束时也可以指定一个退出码，`dwExitCode` 就用来指定进程的退出码。

`ExitProcess` 函数只能用来结束当前进程，不能用于结束其他进程，包括当前进程创建的子进程，因为它并没有参数可以用来输入进程句柄。如果需要结束其他进程的执行，可以使用 `TerminateProcess` 函数：

invoke	TerminateProcess, hProcess, dwExitCode
--------	--

`hProcess` 参数用来指定需要结束的进程的句柄，`dwExitCode` 用来指定进程的退出码。

`TerminateProcess` 函数不是一个推荐使用的函数，一般仅在很极端的情况下使用（如任务管理器用来结束停止响应的进程），因为它将目标进程无条件结束，被结束的进程根本没有机会进行扫尾工作，同时，目标进程使用的 `dll` 文件也不会收到结束通知，所以极有可能造成数据丢失。

当进程被结束的时候，系统将做下面的工作：

- （1）进程创建或打开的所有对象句柄被关闭。
- （2）进程中的所有线程被终止。
- （3）进程及进程中所有线程的状态被改为置位状态，以便让 `WaitForSingleObject` 函数正确检测。
- （4）进程对象中的退出码字段从 `STILL_ACTIVE` 被改为指定的退出码。

大家还记得在 DOS 下编写批处理文件的时候使用的 `ERRORLEVEL` 吗？批处理中可以通过检测 `ERRORLEVEL` 来执行不同的逻辑，这个 `ERRORLEVEL` 就是命令行窗口中上次执行的可执行程序返回的退出码。Win32 中窗口程序的退出码是无法做这个用途了，但它也可以用来在程序退出后向父进程传递简单的状态信息。

要检测进程的退出码，可以使用 `GetExitCodeProcess` 函数：

invoke	GetExitCodeProcess, hProcess, lpExitCode
--------	--

`hProcess` 参数指定被检测进程的进程句柄，`lpExitCode` 指向一个双字变量，用来接收函数返回的退出码。如果执行成功，函数返回非 0 值并将退出码返回到 `lpExitCode` 指定的变量中，如果执行失败函数返回 0。如果被检测的进程没有结束，那么返回到 `lpExitCode` 中的是 `STILL_ACTIVE`。

通过检测子进程的退出码是否是 `STILL_ACTIVE`，就可以得知子进程是否已经结束，但如果在父进程中等待子进程结束时，就没有必要在一个循环中不停地检测退出码。在上一章中介绍的 `WaitForSingleObject` 函数也可以用于等待进程结束，在程序中只要按如下使用就可以了：

invoke	WaitForSingleObject, hProcess, dwMilliseconds
--------	---

如果超时参数 dwMilliseconds 指定 INFINITE，表示在子进程结束前函数不会返回。

当一个进程被结束的时候，并不影响它创建的子进程，进程对象也不会马上从内存中删除，因为可能其他进程还需要通过进程句柄检测进程状态，直到使用 CloseHandle 函数将进程句柄关闭以后，进程对象才真正被删除。所以，当不再需要进程句柄的时候，不要忘记关闭 PROCESS_INFORMATION 结构中返回的进程句柄和主线程句柄。

13.3 进程调试

在 DOS 操作系统下，一个程序可以读写系统中的所有内存，所以可以方便地修改任何地方的代码和数据，不管这些代码和数据是不是自己所有的，另外，程序可以自由存取所有的寄存器，自由设置所有的中断，程序可以通过设置单步中断或断点中断来跟踪代码的执行。这些功能可以归结为对一个进程进行调试。

在 Windows 操作系统中，不同进程之间的地址空间是隔离的，要用指令直接存取其他进程地址空间中的代码和数据是不可能的，用户程序也没有权限去截获中断，通常情况下，甚至连在自己的代码段中写数据都是不合法的，那么在 Windows 中还可以实现类似 DOS 中的调试功能吗？答案是肯定的，但必须通过专用的 API 函数来完成，本节要讨论的就是这方面的内容。

13.3.1 获取运行中的进程句柄

要对进程进行某种操作，就必须首先知道该进程的进程句柄或者进程 ID，否则一切无从谈起，对于程序自己创建的子进程来说，CreateProcess 函数返回了进程句柄和进程 ID，但如果需要调试系统中已经运行的进程，那就必须首先获取它们的句柄才行。

Win32 中并没有直接获取其他进程句柄的函数，但如果知道进程 ID 的话，却可以由此得到进程句柄，所以可以首先通过某种途径获取进程 ID。

1. 从窗口句柄获取进程句柄

获取进程 ID 的方法之一是使用 GetWindowThreadProcessId 函数，这个函数可以从一个窗口句柄获得创建该窗口的进程的进程 ID，而通过 FindWindow 函数得到窗口句柄是很简单的，所以 GetWindowThreadProcessId 函数的用途相当广泛。该函数的用法是：

invoke	GetWindowThreadProcessId, hWnd, lpdwProcessId
--------	---

其中 hWnd 参数指定需要用来获取进程 ID 的窗口句柄，lpdwProcessId 指向一个双字变量，函数在这里返回创建窗口的进程 ID，函数的返回值是目标进程中创建该窗口的线程的线程 ID（一个有用的副产品！）。

得到了进程 ID 以后，就可以通过 OpenProcess 函数来获取该进程的句柄了：

invoke	OpenProcess, dwDesiredAccess, bInheritHandle, dwProcessId
.if	eax

```
        mov     hProcess, eax
    .endif
```

函数的参数定义如下。

- **dwDesiredAccess**——指定需要对该进程进行的操作，要对目标进程进行某种操作，必须指定操作代码，但是在 Windows NT 操作系统中，对其他进程操作需要有相应的权限，如需要结束目标进程就必须有 `PROCESS_TERMINATE` 权限才行，当权限不够的时候，打开进程的操作就会失败。一般来说，除了系统进程以外，可以对其他进程进行任何操作，操作码可以是以下取值的组合：
 - `PROCESS_ALL_ACCESS`——等于下面全部操作码的组合。
 - `PROCESS_CREATE_THREAD`——允许创建远程线程。
 - `PROCESS_DUP_HANDLE`——允许进程句柄被复制。
 - `PROCESS_QUERY_INFORMATION`——允许使用 `GetExitCodeProcess` 函数查询进程的退出码或使用 `GetPriorityClass` 函数查询进程的优先级。
 - `PROCESS_SET_INFORMATION`——允许使用 `SetPriorityClass` 函数设置进程的优先级。
 - `PROCESS_TERMINATE`——允许终止进程。
 - `PROCESS_VM_OPERATION`——允许使用 `WriteProcessMemory` 函数或 `VirtualProtectEx` 函数修改进程的地址空间。
 - `PROCESS_VM_READ`——允许对进程的地址空间进行读操作。
 - `PROCESS_VM_WRITE`——允许对进程的地址空间进行写操作。
- **bInheritHandle**——指明返回的进程句柄是否可以被当前进程的子进程继承，如果参数指定为 `TRUE`，则句柄可以被继承。
- **dwProcessId**——指定目标进程的进程 ID。

如果函数执行成功，返回值是被打开的进程句柄。如果函数执行失败则返回 `NULL`。一般打开失败的原因是由权限不够引起的。当完成对目标进程的操作以后，必须使用 `CloseHandle` 将获得的句柄关闭。

2. 从快照函数获取进程句柄

使用 `GetWindowThreadProcessId` 获取进程 ID 的先决条件是进程必须创建了窗口，对于在后台运行的没有窗口的进程该如何处理呢？这就要通过枚举系统中运行的进程来解决，这个功能可以由 `CreateToolhelp32Snapshot` 函数来实现。

通过 `CreateToolhelp32Snapshot` 函数可以获得一个进程的列表，可以从列表中得到进程的 ID、进程对应的可执行文件名和创建该进程的父进程 ID 等数据，这个函数支持 Windows 9x 系列和 Windows 2000 及以上的系统，不支持 Windows NT 4.0（幸好使用 NT 4.0 的机会已经不多了）。

哭

图 13.4 “快照”例子的运行界面

目录中的 ProcessList.rc 文件定义了如图 13.4 所示的对话框。

[illegible]

汇编源文件 ProcessList.asm 的内容如下:

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include        windows.inc
include        user32.inc

```

```

include      user32. lib
include      kernel32. inc
includelib   kernel32. lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Equ 等值定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ICO_MAIN equ    1000
DLG_MAIN equ    1000
IDC_PROCESS equ    1001
IDC_REFRESH equ    1002
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .data?
hInstancedd    ?
hWinList dd    ?

                .const
szErrTerminate db    ' 无法结束指定进程!', 0
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 代码段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                .code
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_GetProcessList proc    _hWnd
                local    @stProcess:PROCESSENTRY32
                local    @hSnapshot

                invoke    RtlZeroMemory, addr @stProcess, sizeof @stProcess
                invoke    SendMessage, hWinList, LB_RESETCONTENT, 0, 0
                mov       @stProcess. dwSize, sizeof @stProcess
                invoke    CreateToolhelp32Snapshot, TH32CS_SNAPPROCESS, 0
                mov       @hSnapshot, eax
                invoke    Process32First, @hSnapshot, addr @stProcess
                .while    eax
                invoke    SendMessage, hWinList, LB_ADDSTRING, \
                        0, addr @stProcess. szExeFile
                invoke    SendMessage, hWinList, LB_SEITEMDATA, eax, \
                        @stProcess. th32ProcessID
                invoke    Process32Next, @hSnapshot, addr @stProcess
                .endw
                invoke    CloseHandle, @hSnapshot
                invoke    GetDlgItem, _hWnd, IDOK
                invoke    EnableWindow, eax, FALSE
                ret
_GetProcessList endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_ProcDlgMain    proc    uses ebx edi esi hWnd, wParam, lParam

                mov       eax, wParam
                .if       eax == WM_CLOSE
                invoke    EndDialog, hWnd, NULL
                .elseif   eax == WM_INITDIALOG

```



```

        invoke    GetDlgItem, hWnd, IDC_PROCESS
        mov      hWinList, eax
        invoke    _GetProcessList, hWnd
;*****
        .elseif  eax == WM_COMMAND
        mov      eax, wParam
        .if      ax == IDOK
            invoke    SendMessage, hWinList, LB_GETCURSEL, 0, 0
            invoke    SendMessage, hWinList, \
                LB_GETITEMDATA, eax, 0
            invoke    OpenProcess, PROCESS_TERMINATE, \
                FALSE, eax
        .if      eax
            mov      ebx, eax
            invoke    TerminateProcess, ebx, -1
            invoke    CloseHandle, ebx
            invoke    Sleep, 200
            invoke    _GetProcessList, hWnd
            jmp      @F
        .endif
        invoke    MessageBox, hWnd, addr szErrTerminate, \
            NULL, MB_OK or MB_ICONWARNING
        @@:
;*****
        .elseif  ax == IDC_REFRESH
            invoke    _GetProcessList, hWnd
;*****
        .elseif  ax == IDC_PROCESS
            shr      eax, 16
            .if      ax == LBN_SELCHANGE
                invoke    GetDlgItem, hWnd, IDOK
                invoke    EnableWindow, eax, TRUE
            .endif
        .endif
;*****
        .else
            mov      eax, FALSE
            ret
        .endif
        mov      eax, TRUE
        ret

_ProcDlgMain    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
start:
        invoke    GetModuleHandle, NULL
        mov      hInstance, eax
        invoke    DialogBoxParam, hInstance, DLG_MAIN, \
            NULL, offset _ProcDlgMain, NULL
        invoke    ExitProcess, NULL
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        end      start

```

程序在初始化、按下“刷新”按钮，以及结束一个线程以后都要调用_GetProcessList 子程序来重新获取系统中运行的进程列表。在_GetProcessList 子程序中，程序首先向列表框发送 LB_RESETCONTENT 消息删除原来的列表内容，并调用 CreateToolhelp32Snapshot 函数获得一个快照。函数的使用格式是：

```

invoke    CreateToolhelp32Snapshot, dwFlags, th32ProcessID
.if      eax
        mov     hSnapShot, eax
.endif

```

dwFlags 参数用来指定“快照”中需要返回的对象，本函数不仅可以获取进程列表，也可以用来获取线程和模块等对象的列表，参数可以指定的值是：

- TH32CS_SNAPHEAPLIST——对指定进程中的堆进行枚举。
- TH32CS_SNAPMODULE——对指定进程中的模块进行枚举。
- TH32CS_SNAPPROCESS——对系统范围中的进程进行枚举。
- TH32CS_SNAPTHREAD——对系统范围中的线程进行枚举。

th32ProcessID 参数用来指定一个进程 ID，当 dwFlags 指定为 TH32CS_SNAPHEAPLIST 或者 TH32CS_SNAPMODULE 来枚举某个进程中的堆，以及模块的时候，这个参数用来指定被枚举的进程 ID。对于 TH32CS_SNAPPROCESS 和 TH32CS_SNAPTHREAD 标志，由于枚举的范围是系统范围内的，所以 th32ProcessID 参数将被忽略。

如果函数执行成功，将返回一个快照句柄，否则返回-1。程序可以通过这个快照句柄获取进程列表。

从快照句柄中获取进程参数使用 Process32First 和 Process32Next 函数，函数的每次调用仅返回一个进程的信息。Process32First 函数用来进行首次调用，以后的调用由 Process32Next 函数循环完成，直到所有的进程信息都被获取为止，当不再有剩余信息的时候，函数返回 FALSE，所以一般使用下面的循环结构来获取进程列表：

```

                .data?
stProcess      PROCSENTRY32<?>
hSnapShot      dd      ?
                .code
invoke         CreateToolhelp32Snapshot, TH32CS_SNAPPROCESS, 0
mov           hSnapShot, eax
mov           stProcess.dwSize, sizeof stProcess
invoke        Process32First, hSnapShot, addr stProcess
.while        eax
                ;在这里处理返回到 PROCSENTRY32 中的进程信息
                invoke     Process32Next, hSnapShot, addr stProcess
.endif
invoke        CloseHandle, hSnapShot

```

Process32First 和 Process32Next 函数的第一个参数是前面得到的快照句柄，第二个参数指向一个 PROCSENTRY32 结构，进程信息将被返回到这个结构中。结构的定义如下：

PROCESSENTRY32 STRUCT

dwSize	DWORD ?	;结构的长度, 必须预先设置
cntUsage	DWORD ?	;进程的引用计数
th32ProcessID	DWORD ?	;进程 ID
th32DefaultHeapID	DWORD ?	;进程默认堆的 ID
th32ModuleID	DWORD ?	;进程模块的 ID
cntThreads	DWORD ?	;被进程创建的线程数
th32ParentProcessID	DWORD ?	;进程的父进程 ID
pcPriClassBase	DWORD ?	;被进程创建的线程的基本优先级
dwFlags	DWORD ?	;内部使用
szExeFile	db MAX_PATH dup(?)	;进程对应的可执行文件名

PROCESSENTRY32 ENDS

注意: 在使用前需要先将 dwSize 填写为结构的长度, 否则函数的执行会失败, 在返回所有的进程信息后, 需要使用 CloseHandle 函数将快照句柄关闭。

结构中返回的进程 ID 字段 (th32ProcessID) 和可执行文件名字段 (szExeFile) 是我们最关心的。通过比较文件名, 就可以找出需要寻找的可执行文件产生的进程, 然后通过进程 ID 就可以用 OpenProcess 函数获得进程句柄, 有了进程句柄以后就可以对进程进行各种操作了。

在例子中, 每当在 PROCESSENTRY32 结构中返回了一个进程的信息后, 程序向列表框发送 LB_ADDSTRING 消息将可执行文件名添加到列表框中。由于列表框能够为每个项目定义一个 32 位的自定义数据, 利用这个特征可将文件对应的进程 ID 保存到这里, 方法就是向列表框发送 LB_SETITEMDATA 消息, 这样在按下“终止”按钮以后, 程序就可以通过 LB_GETITEMDATA 消息取回进程 ID, 使用 OpenProcess 函数获得该进程的句柄以后, 再使用 TerminateProcess 函数将进程终止。

当例子程序在 Windows 2000 中运行时, 如果试图打开系统底层的进程 (如 System 或者 [System Process] 等进程) 是不会成功的, 因为用户程序并没有这么高的权限。

在详细介绍了枚举系统中所有进程的方法后, 下面再附带介绍一下枚举堆、线程和模块的方法, 换句话说, 就是 CreateToolhelp32Snapshot 函数中使用了 TH32CS_SNAPHEAPLIST、TH32CS_SNAPMODULE、TH32CS_SNAPTHREAD 参数后, 如何再得到堆、线程和模块列表的问题。

难道是同样使用 Process32First 和 Process32Next 函数构成一个循环吗? 答案是否定的, 实际上, 循环的结构是同样的, 但构成循环的函数要分别换成 Heap32First 和 Heap32Next, Thread32First 和 Thread32Next, 另外还有 Module32First 和 Module32Next。用来返回数据的结构也是不同的, 它们分别是 HEAPENTRY32、THREADENTRY32 和 MODULEENTRY32。关于这些函数和结构的资料, 读者可以自己从 MSDN 中查看相关的资料。

13.3.2 读写进程的地址空间

1. 进程地址空间的读写函数

当一个进程能够被我们以足够的权限打开以后, 就可以通过 ReadProcessMemory 和 WriteProcessMemory 函数读写它的地址空间。只要能够对其他进程的地址空间进行读写, 那么

我们能够做的事情就多了，只要发挥想象力，就能够编出一些超乎想象的程序来。在广为流传的应用程序中，最为著名的就是 FPE 之类的游戏修改器。

FPE 是 Fix People Expert 的缩写，不过这个软件可不是用来修理人（Fix People）而是用来对付游戏的。FPE 程序列出当前系统中运行的进程，让用户选择要对付的游戏程序名（现在读者可以骄傲地说，这一招我也会，13.3.1 节中的进程列表例子不就是这样吗），然后让用户输入一个数值，比如，现在游戏主角还剩下 3 条命就输入 3，FPE 将扫描游戏进程的所有地址空间，将当前数据为 3 的地址列入黑名单，接下来继续游戏，当又少了一条命的时候，再次输入 2 并扫描，如果上次黑名单中某个地址中的数据现在变成了 2，代表生命的数据十有八九就存放在这个地址中，将它改成 100 的话，主角就长命百岁了！同样道理想让主角变成千千岁，万万岁也不在话下！另外，如果找到代表生命的地址，让 FPE 锁定（就是每隔很短的时间将要锁定的数值重新写到这个地址中）数值，游戏主角就是金刚不坏之躯了。

FPE 是通用的修改软件，另一类专用的游戏修改器也使用同样的技术，比如，打《暗黑破坏神》游戏的时候，很多人用过增加经验点数的修改器，因为这个游戏对数据经过了某种处理，用 FPE 一类的软件无法直接将要修改的数据搜索出来，有人就通过跟踪找到了变换后的数据地址和变换算法，并专门写了针对这个游戏的进程内存读写程序。

当然，经过本节介绍以后，读者就会觉得这些软件使用的技术并没有那么神秘，我们自己也可以写出同样的程序来。

首先来介绍一下这两个函数的用法：

invoke	ReadProcessMemory, hProcess, lpBaseAddress, lpBuffer, \	
	dwSize, lpNumberOfBytesRead	(读进程内存)
invoke	WriteProcessMemory, hProcess, lpBaseAddress, lpBuffer, \	
	dwSize, lpNumberOfBytesWritten	(写进程内存)

这两个函数的参数定义是一样的，各参数的定义如下：

- hProcess——指定将要被读写的目标进程句柄。
- lpBaseAddress——目标进程中被读写的起始线性地址。
- lpBuffer——用来接收读取数据的缓冲区（对于 ReadProcessMemory 函数）或者要写到目标进程的数据缓冲区（对于 WriteProcessMemory 函数）。
- dwSize——要读写的字节数。
- lpNumberOfBytesRead 或 lpNumberOfBytesWritten——指向一个双字变量，用来供函数返回实际读写的字节数，如果不关心读写的结果，可以在这里使用 NULL。

如果函数执行成功，那么返回值是非 0 值，执行失败的话返回 0。

使用这两个函数需要注意的地方有：

- 注意 lpBaseAddress 和 lpBuffer 参数指向的地址位于不同的进程空间内，lpBuffer 指向的缓冲区位于本进程的地址空间内，而 lpBaseAddress 指向的地址位于目标进程的地址空间内。

- 要对目标进程进行读写的话，打开（或者创建）目标进程的时候必须包含对应的权限，如要读取目标进程必须包括 `PROCESS_VM_READ` 权限；要对目标进程进行写操作的时候，必须包括 `PROCESS_VM_OPERATION` 或者 `PROCESS_VM_WRITE` 权限。
- `lpBaseAddress` 位置开始的 `dwSize` 大小的内存必须是可存取的，函数在执行前会对整个区域进行测试，如果中间有某处是不可存取的（如没有被递交到物理内存），那么函数直接返回失败，所以一般不会出现只读写了一部分内存的情况。
- 虽然在自己的进程中，代码段是不可写的，但是使用 `WriteProcessMemory` 函数去写目标进程的代码段却是允许的。

FPE 等软件就是使用这两个函数来读写目标进程的数据的，但由于这些函数也可以用来读写目标进程的代码部分，所以也有一些程序使用它们来做内存补丁，下面介绍的一个例子演示了如何利用该函数来修改代码。

2. 一个内存补丁例子

与这个例子相关的文件放在所附光盘的 `Chapter13\Patch1` 目录中，其中 `Test.asm` 是将被修改的目标程序，这个程序中只有几句代码：

```

        .const
szErr    db      '对不起，你使用的是盗版软件!',0
szOK     db      '感谢您使用正版软件!',0
szCaption db      '谢谢',0
        ...
        .code
start:
        xor      eax,eax
        .if      eax
            invoke MessageBox,NULL,addr szOK,\
                addr szCaption,MB_OK
        .else
            invoke MessageBox,NULL,addr szErr,NULL,\
                MB_OK or MB_ICONSTOP
        .endif
        ...

```

在程序中模拟测试序列号的过程，当 `eax` 返回 0 的时候，显示“对不起，你使用的是盗版软件”，否则显示“感谢您使用正版软件”，为了简化程序，程序用了一句 `xor eax, eax` 指令使 `eax` 永远为 0，所以显示的总是“盗版软件……”。我们的目标就是编写一个内存补丁程序，让它去调用 `Test.exe` 并修改测试代码，这样就可以跳过“反盗版”测试。

要对某个软件进行内存补丁，就必须首先对它进行分析，这样才能知道该往目标进程的什么地方写入什么数据。这方面的工作属于软件跟踪的课题，在本书中暂不涉及，有兴趣的读者可以另外研究。对于上面这个简单的 `Test.exe` 文件，采用静态分析的方法，用 `W32Dasm` 将它反汇编，就可以得到下面的代码（在实际的使用中，就是仁者见仁、智者见智了，读者可以用自己擅长的方法分析目标文件）：

```

:00401000 33C0          xor     eax, eax

```

```
:00401002 0BC0          or eax, eax                ;.if eax
:00401004 7415          je 0040101B              ;<---- 修改这条指令
:00401006 6A00          push 00000000
:00401008 6840204000   push 00402040
:0040100D 682C204000   push 0040202C
:00401012 6A00          push 00000000
:00401014 E819000000   call 00401032            ; “正版” MessageBox
:00401019 EB10          jmp 0040102B
:0040101B 6A10          push 00000010           ;.else
:0040101D 6A00          push 00000000
:0040101F 6810204000   push 00402010
:00401024 6A00          push 00000000
:00401026 E807000000   call 00401032           ;MessageBox
```

分析这段代码就可以发现，只要将 00401004h 地址的 je 指令略过就可以让程序执行“正版”逻辑，在内存补丁程序中使用两个 nop 指令代替 je 指令，nop 指令的机器码为 90h，为单字节指令，而 je 指令有两个字节，所以需要两个 nop 指令。

读者可能还有一个问题：反汇编的时候显示地址是 00401004h，难道执行的时候实际装入的地址也是这个吗？对于 exe 程序来说，这是肯定的，因为不同进程的地址空间是隔离的，不会有其他东西占用这部分地址；对于 DLL 来说就不一定了，因为一个 exe 文件可能装入多个 DLL，当两个 DLL 的默认装入地址相同时，有一个肯定会被重新定位到其他地方，所以对 DLL 进行静态反汇编得到的地址不一定是正确的，在这种情况下可以扫描目标进程的整个空间来找到 DLL 的实际装入位置（这方面的知识请参考第 17 章：PE 文件）。

分析目标代码得到要补丁的地址和该地址处的新旧机器码以后,就可以写出下面的补丁程序了,程序的汇编源文件为 Patch1.asm:

```

        .386
        .model flat, stdcall
        option casemap :none    ; case sensitive
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;                Include 数据
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include        windows.inc
include        user32.inc
include        kernel32.inc
includelib     user32.lib
includelib     kernel32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
PATCH_POSITION    equ        00401004h
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
;                数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .data?
dbOldBytes        db          2 dup (?)
stStartUpSTARTUPINFO    <?>
stProcInfo        PROCESS_INFORMATION    <?>

        .const
dbPatch          db          74h, 15h

```


能将补丁程序用在不同版本的目标程序中，这时候补丁就会写入错误的地址，这可能会引起目标程序崩溃，首先验证补丁处的原有内容是否正确，就可以防止发生这种错误的发生。通过了内容的校验以后，程序再使用 WriteProcessMemory 函数将补丁写入正确的位置。

虽然程序很简单，但是读者还要注意两个细节：首先是创建进程的时候最好使用 CREATE_SUSPENDED 标志，这样目标进程的主线程一开始是被挂起的，可以防止补丁程序在执行补丁代码的过程中被 Windows 打断，然后切换到目标进程去的情况，当补丁完成以后，使用 ResumeThread 函数恢复目标进程的执行就可以了；第二点是完成补丁以后，不要忘记使用 CloseHandle 语句将 CreateProcess 返回的进程句柄和线程句柄关闭。

例子中使用了 CreateProcess 函数来创建目标进程，对于使用这种方法得到的目标进程句柄，父进程拥有全部权限，可以自由读写子进程的地址空间。如果使用 OpenProcess 去获取运行中的目标进程的句柄时，不要忘了指定 PROCESS_VM_READ 和 PROCESS_VM_WRITE 权限，否则补丁操作会失败。

13.3.3 调试API 的使用

对于上面的内存补丁例子，读者可能会说：这种东西还好意思拿出来亮相，又没有实用价值！只要用十六进制编辑器去 Test.exe 中查找数据串“74h、15h”然后直接改成“90h、90h”就一劳永逸了，何必这样兴师动众写一个补丁程序呢？的确如此，不过使用这个例子是为了介绍 ReadProcessMemory 和 WriteProcessMemory 函数的用法，而且可以用来继续引出本节中的例子程序。

稍微接触过加密解密的读者肯定见过“加壳程序”和“脱壳程序”这两个名词。“加壳”指将可执行文件的代码和数据经过某种变换后存储，并在原来的可执行文件中添加一段用于还原的代码，这样在执行程序的时候，这些代码会自动将原来可执行文件的代码和数据还原并执行，用户并不会感觉到程序被改动过，这段用于还原的代码就像是一层壳附在原来文件外面，所以对文件进行变换处理的程序就被叫做“加壳程序”。

“加壳”的原因有两个：压缩和加密。压缩程序可以将可执行文件的内容压缩存储，这样文件占用的磁盘空间可以缩小，这时“壳代码”就是解压缩代码；而加密程序则是为了保证可执行文件的内容不被随便修改（就像上面讲的用十六进制编辑器去修改关键代码），这时“壳代码”就是解密代码，一般解密代码中同时包含有反跟踪模块。现在的大部分加密软件同时有压缩的功能，如 Aspack, PECompact 和 ASProtect 等软件。在被加壳的文件中，原来的代码和数据已经面目全非了，用十六进制编辑器去寻找特征码是根本无法找到的，所以无法用修改文件的方法进行静态补丁。

要对加过壳的软件进行修改必须首先将它脱壳，但大部分的加壳软件都无法用简单的方法去对付，除了一些加密程度不高的“壳”可以利用逆算法完全恢复原来的文件外，有很多“壳”是无法用逆运算对付的。虽然可以用 Soft-ICE 等跟踪软件跟踪到壳代码的内部，并一直跟踪到壳代码将原来的可执行文件恢复为止，然后再手工去改动内存中的关键代码，但用户不可能为了执行程序而每次都用调试器去载入可执行文件，并且花一段时间去跟踪并做“手工补丁”。

[illegible]

程序不长，不到 100 行，这在 Win32 汇编程序中的规模已经是相当小了，但由于调试 API 使用的数据结构比较复杂，所以代码看起来比较费解，如下面的代码：

读者看到这里千万不要打退堂鼓，这只是 4 层结构的嵌套而已，接下来就会介绍它们的使用方法。（不过退一步讲，如果调试 API 使用起来很简单，Soft-ICE 之类的软件就显示不出它们的伟大了！）

在其他的一些情况中，也许会遇到调试不是由程序本身创建的子进程的情况，这时可以通过 `DebugActiveProcess` 函数让目标进程进入调试状态：

dwProcessId 参数指定需要调试的进程 ID。

2. 调试 API

在用做调试器的父进程中，可以通过调试 API 完成下面的功能，如果愿意的话，可以使用这些功能来写一个全功能的 Debug 程序：

- 获取被调试程序的底层信息，如进程 ID、入口地址和映象基址等。
- 当发生与调试有关的事件时获得通知，如进程或线程的开始和结束，DLL 的加载和释放，以及发生各种异常等。
- 修改被调试的进程或线程。

当被调试进程发生与调试有关的事件时，Windows 将目标进程的线程挂起并给调试器发送一个事件通知，调试器进程使用 WaitForDebugEvent 函数获取这些事件，在处理完调试事件后，可以恢复目标进程的执行并等待下一个事件的发生，例子程序就是在这样一个无限循环中使用 WaitForDebugEvent 函数来获取调试事件并处理的。

WaitForDebugEvent 函数的用法是：

invoke	WaitForDebugEvent, lpDebugEvent, dwMilliseconds
--------	---

dwMilliseconds 指定一个以 ms 为单位的等待时间。如果要一直等待到某个事件发生函数才返回，可以在这里使用 INFINITE 值。lpDebugEvent 参数指向一个 DEBUG_EVENT 结构，函数在这里返回调试事件的具体信息。DEBUG_EVENT 结构的定义比较复杂，例子代码看上去很复杂的原因就是因为这个结构的复杂性。结构的定义为：

DEBUG_EVENT STRUCT			
dwDebugEventCode	DWORD	?	; 调试事件类型
dwProcessId	DWORD	?	; 发生调试事件的进程 ID
dwThreadId	DWORD	?	; 发生调试事件的线程 ID
u	DEBUGSTRUCT <>		; 事件的具体信息
DEBUG_EVENT ENDS			

dwDebugEventCode 字段指定了发生的调试事件类型。因为可能发生的事件类型有很多种，所以程序要检查此字段并根据不同的事件做不同的处理。例子程序中设置了一个逻辑分支代码来处理 EXIT_PROCESS_DEBUG_EVENT，CREATE_PROCESS_DEBUG_EVENT 以及 EXCEPTION_DEBUG_EVENT 事件。可能发生的事件有：

- CREATE_PROCESS_DEBUG_EVENT——进程被创建。当使用 CreateProcess 函数刚创建被调试进程（还未开始运行），或者已经运行中的进程刚被 DebugActiveProcess 函数捆绑到调试器中时发生这个事件。
- EXIT_PROCESS_DEBUG_EVENT——被调试进程退出。
- CREATE_THREAD_DEBUG_EVENT——被调试进程中新建了一个线程（但是被调试进程的主线程被创建的时候不会收到这个事件）。
- EXIT_THREAD_DEBUG_EVENT——被调试进程中某个线程结束。

- `LOAD_DLL_DEBUG_EVENT`——被调试进程装入一个 DLL 时发生本事件。当系统分析可执行文件并根据文件头中的导入表装入 DLL 时，程序会收到这个事件；当被调试进程使用 `LoadLibrary` 装入 DLL 时也会发生本事件。
- `UNLOAD_DLL_DEBUG_EVENT`——当一个 DLL 从被调试进程中卸载时发生本事件。
- `EXCEPTION_DEBUG_EVENT`——被调试进程中发生异常事件。被调试进程开始执行第一条指令前本事件会发生一次，以后只有在发生调试中断（遇到 `int 3` 或者单步中断），以及发生异常的时候才会发生本事件。
- `OUTPUT_DEBUG_STRING_EVENT`——当被调试进程调用 `DebugOutputString` 函数时发生本事件。被调试进程可以通过这种方法向调试器程序发送消息字符串。
- `RIP_EVENT`——系统调试发生错误。

`DEBUG_EVENT` 结构中的 `dwProcessId` 和 `dwThreadId` 字段为发生调试事件的进程和线程 ID。虽然使用 `CreateProcess` 创建被调试进程的时候就已经得到两个 ID 值，但这两个 ID 是属于子进程的。当没有指定 `DEBUG_ONLY_THIS_PROCESS` 标志时，调试事件可能发生在“孙”进程中，这时 `dwProcessId` 和 `dwThreadId` 字段指明的就是“孙”进程的 ID。通过检测这个 ID 值和调用 `CreateProcess` 函数时获取的 ID 值是否一致，可以知道事件是发生在子进程还是孙进程中。

`u` 字段包含了调试事件的详细信息，根据 `dwDebugEventCode` 的不同，它被定义为不同的结构，结构的名称和事件的对应关系如表 13.1 所示。在这里由于篇幅的关系就不列出所有的结构定义了，读者可以具体参考 Win32 API 手册。

表 13.1 发生不同调试事件时 `u` 字段中的结构

<code>dwDebugEventCode</code>	<code>u</code> 字段中的结构
<code>CREATE_PROCESS_DEBUG_EVENT</code>	<code>CREATE_PROCESS_DEBUG_INFO</code>
<code>EXIT_PROCESS_DEBUG_EVENT</code>	<code>EXIT_PROCESS_DEBUG_INFO</code>
<code>CREATE_THREAD_DEBUG_EVENT</code>	<code>CREATE_THREAD_DEBUG_INFO</code>
<code>EXIT_THREAD_DEBUG_EVENT</code>	<code>EXIT_THREAD_DEBUG_EVENT</code>
<code>LOAD_DLL_DEBUG_EVENT</code>	<code>LOAD_DLL_DEBUG_INFO</code>
<code>UNLOAD_DLL_DEBUG_EVENT</code>	<code>UNLOAD_DLL_DEBUG_INFO</code>
<code>EXCEPTION_DEBUG_EVENT</code>	<code>EXCEPTION_DEBUG_INFO</code>
<code>OUTPUT_DEBUG_STRING_EVENT</code>	<code>OUTPUT_DEBUG_STRING_INFO</code>
<code>RIP_EVENT</code>	<code>RIP_INFO</code>

当程序使用 `WaitForDebugEvent` 函数获取了一个事件并进行处理以后，被调试进程还处在挂起状态，调试事件处理完毕后让它恢复运行是调试器的责任，恢复被调试进程的运行可以使用 `ContinueDebugEvent` 函数：

```
invoke ContinueDebugEvent, dwProcessId, dwThreadId, dwContinueStatus
```

其中，dwProcessId 和 dwThreadId 参数指定被恢复运行的进程 ID 和线程 ID，在这里可以直接使用在 DEBUG_EVENT 结构中返回的同名字段。dwContinueStatus 参数指定恢复运行的方式，一般指定为 DBG_CONTINUE。

总之，使用下面的循环结构进行调试过程：

```
.while TRUE
    invoke WaitForDebugEvent, addr DebugEvent, INFINITE
    .break .if DebugEvent.dwDebugEventCode==EXIT_PROCESS_DEBUG_EVENT
    .if DebugEvent.dwDebugEventCode==XXXXXXX
        <处理调试事件 1>
    .elseif DebugEvent.dwDebugEventCode==YYYYYYY
        <处理调试事件 2>
    ...
    .endif
    invoke ContinueDebugEvent, DebugEvent.dwProcessId, \
        DebugEvent.dwThreadId, DBG_CONTINUE
.endw
```

循环中首先用 WaitForDebugEvent 函数获取调试事件，然后用一个分支语句检测事件类型，当发现事件代码为 EXIT_PROCESS_DEBUG_EVENT（进程退出）时，用 .break 语句结束循环；在其他情况下，程序根据不同的事件码进行不同的处理。

例子程序中处理了 CREATE_PROCESS_DEBUG_EVENT 和 EXCEPTION_DEBUG_EVENT 事件。当发生 CREATE_PROCESS_DEBUG_EVENT 事件时，表示建立了被调试进程，这时例子程序在目标进程的入口代码处（地址为 00405120h，原指令为 pushad，机器码为 60h）写入一个 0cch（int 3 的机器码），当目标进程开始执行时，Windows 就会以 EXCEPTION_DEBUG_EVENT（异常事件）通知程序。

由于已经在静态反汇编分析中知道了该地址原来的内容，所以将 int 3 指令写入之前不必使用 ReadProcessMemory 函数先保存原来的指令；如果不知道被覆盖的指令码是什么，那么在写入 int 3 之前就必须保存原来的指令码，因为以后还要将它恢复回去。

接下来就是等待这个 int 3 发生了，也就是 EXCEPTION_DEBUG_EVENT 事件的发生，对于 EXCEPTION_DEBUG_EVENT 事件，u 字段被定义为 EXCEPTION_DEBUG_INFO 结构：

```
EXCEPTION_DEBUG_INFO STRUCT
    pExceptionRecord EXCEPTION_RECORD <?, ?, ?, ?, EXCEPTION_MAXIMUM_PARAMETERS dup(?)>
    dwFirstChance      DWORD      ?
EXCEPTION_DEBUG_INFO ENDS
```

结构中的 pExceptionRecord 字段又被定义为一个 EXCEPTION_RECORD 结构，在这个结构中，我们需要的信息才浮出水面：

```
EXCEPTION_RECORD STRUCT
    ExceptionCode      DWORD      ?      ;异常事件码
    ExceptionFlags      DWORD      ?      ;标志
    pExceptionRecord    DWORD      ?
    ExceptionAddress     DWORD      ?
    NumberParameters    DWORD      ?
    ExceptionInformation DWORD      EXCEPTION_MAXIMUM_PARAMETERS dup(?)
```

EXCEPTION_RECORD ENDS

读者不要被嵌套得这么深的结构吓倒了，实际上对它们的引用很简单，只要使用“结构 1. 结构 2. 结构 3. 结构 4. 字段 n”的格式就可以了。EXCEPTION_RECORD 结构的 ExceptionCode 字段定义了异常事件的类型，异常事件的类型有很多，我们关心的是 EXCEPTION_BREAKPOINT（断点中断）和 EXCEPTION_SINGLE_STEP（单步中断）两种异常。好了，现在程序可以在异常事件的处理中再设置一个分支，并根据断点中断和单步中断两种情况做不同的处理。

在分析例子程序对这两种异常事件的处理代码之前，还需要了解一个新概念，即线程环境。

3. 线程环境

在第 12 章中已经提到过，Windows 为不同的线程循环分配时间片，当挂起一个线程的时候，为了以后能够将它恢复执行，系统必须首先将线程的运行环境保存下来，当线程在下一个时间片恢复执行时，将运行环境恢复回去，线程就不会感觉到自己被打断过，这就像甲外出时把办公室交给乙管，不管乙把办公室搞成什么样子，只要在甲回来之前把所有东西恢复原状，甲就不会意识到甲出去的时候办公室被挪做它用了。

线程环境就是这个道理，Windows 中将线程环境称为“Thread Context”（注意：没有进程 Context，因为进程是不活动的），对一个线程来说，只要所有的寄存器没有改变，环境就没有改变，所以线程环境实际上就是寄存器的状态，它可以用一个 CONTEXT 结构来表示。结构的定义如下：

```
CONTEXT STRUCT
    ContextFlags      DWORD      ?
    iDr0              DWORD      ?
    iDr1              DWORD      ?
    iDr2              DWORD      ?
    iDr3              DWORD      ?
    iDr6              DWORD      ?
    iDr7              DWORD      ?
    FloatSave         FLOATING_SAVE_AREA <>
    regGs             DWORD      ?
    regFs             DWORD      ?
    regEs             DWORD      ?
    regDs             DWORD      ?
    regEdi            DWORD      ?
    regEsi            DWORD      ?
    regEbx            DWORD      ?
    regEdx            DWORD      ?
    regEcx            DWORD      ?
    regEax            DWORD      ?
    regEbp            DWORD      ?
    regEip            DWORD      ?
    regCs             DWORD      ?
    regFlag           DWORD      ?
    regEsp            DWORD      ?
    regSs             DWORD      ?
    ExtendedRegisters db MAXIMUM_SUPPORTED_EXTENSION dup(?)
CONTEXT ENDS
```

结构中的字段包括 80x86 系列处理器中的全部寄存器，其中 FloatSave 字段用来保存浮点寄存器的内容，ExtendedRegisters 字段用来保存扩展寄存器的内容（如 MMX 寄存器等），ContextFlags 字段是供结构自己用的标志。

CONTEXT 结构是 Windows 中惟一与硬件平台相关的结构，因为 Windows 设计成可以在不同的硬件平台上运行，当运行于 MIPS, Alpha 和 PowerPC 等平台上时，显然寄存器名称就和 80x86 系列的不同了，这时 CONTEXT 结构的定义也相应改变了。

在线程处于休眠状态的时候，它的线程环境由 Windows 保存，其他程序可以通过 API 获取它们并修改它们，当线程分配到时间片恢复运行时，Windows 将修改过的线程环境恢复回去，而线程并不会意识到环境已经被修改了。用这种方法可以修改 regEip 字段，让某个线程转移到其他地方执行。

用于获取和重新设置线程环境的函数是 GetThreadContext 和 SetThreadContext。有了这两个函数，调试手段中就多了一种利器，想一想，能够随意修改目标线程的内容，也可以随意修改它的运行环境，还有什么事情做不到呢？

这两个函数的用法是：

invoke	GetThreadContext, hThread, lpContext
invoke	SetThreadContext, hThread, lpContext

hThread 指定目标线程句柄，lpContext 指向一个 CONTEXT 结构，GetThreadContext 函数会将目标线程的环境返回到这个结构中，SetThreadContext 函数将结构中的寄存器设置到目标线程中。为了执行这两个函数，程序必须对目标线程拥有 THREAD_GET_CONTEXT 和 THREAD_SET_CONTEXT 权限。

在执行函数前，必须设置 CONTEXT 结构中的 ContextFlags 字段，这个字段表示需要操作的寄存器的范围。访问通用寄存器可以指定 CONTEXT_INTEGER；访问段寄存器可以指定 CONTEXT_SEGMENTS；要访问全部寄存器则指定为 CONTEXT_FULL。

另外，在定义 CONTEXT 结构的时候，应该将它定义为双字对齐，否则，在 NT 下将得到奇怪的结果，双字对齐的方法是在结构的定义前加上“align dword”关键字。

在执行 GetThreadContext 函数前，最好使用 SuspendThread 函数将目标线程挂起，防止函数执行到一半的时候被 Windows 切换走了，在执行 SetThreadContext 以后可以再使用 ResumeThread 函数将目标线程恢复运行。但是在调试事件中就没有这个必要了，因为在 ContinueDebugEvent 函数返回之前，目标线程不会恢复运行。

现在回过头来看例子程序，在发生异常事件的逻辑分支中，当检测到断点中断的时候，程序使用 GetThreadContext 函数获取线程环境，并比较 eip 是否到达我们设置的断点+1 处（因为断点指令执行以后才发生异常，这时候 eip 已经指向了下一条指令），如果到达，则用 WriteProcessMemory 函数将断点处原来的代码恢复回去，并将 eip 寄存器减 1 以便目标线程能够重新执行这条指令，同时，程序将 eflags 标志的单步标志置 1，这样以后每执行一条指令，就会发生单步中断回到调试器中，就可以一步步跟踪，直到壳代码将 Test.exe 的内容完全解

压缩为止。完成了对线程环境的修改以后，使用 SetThreadContext 函数将线程环境设置回去，现在就等下一次的单步中断发生了。

当单步中断发生的时候，程序同样使用 GetThreadContext 函数获取线程环境，并比较 eip 是否到达 Test.exe 程序原来的入口处（00401000h），如果没有到达，程序继续设置单步标志，这是因为 Windows 执行一条语句以后会自动清除单步标志，为了继续进行单步跟踪，必须每次重新设置单步标志；如果发现已经执行到 00401000h 处了，程序开始打内存补丁，这时程序不必再设置单步标志，因为要做的工作全部完成了！

运行程序，在经过几秒的等待后，“感谢您使用正版软件”的消息框终于出现了，这就意味着我们成功地突破了壳代码的封锁。

但是单步中断的开销是很大的，毕竟在原来的一条指令之间要多执行成千上万条指令，效率也就相应低了很多，这就是前面要等待几秒的原因。实际上有时候可以不用单步中断，而采用分步进行断点中断的办法；有时候，一个合适的断点就可以解决全部问题，比如对于用做例子的 Test.exe 程序来说，观察反汇编后的代码：

:0040526E 61	popad
:0040526F E98CBDFFFF	jmp 00401000

可以在 0040526Eh 处设置断点，这样就可以在壳代码完成解压缩操作以后再将它中断，而且只要中断一次就可以了，按这种方法编写的补丁程序放在所附光盘的 Chapter13\Patch3 目录中，这个程序执行起来没有一点延时，有兴趣的读者自己分析一下。实际上，Patch2 程序舍近求远用了单步中断的原因仅是为了给读者做处理单步中断的示范。

当然，使用调试 API 编写补丁程序的时候，采用的方案是建立在对目标程序的分析上的，并没有一个很确定的方案，本节的例子给出了 Patch1 到 Patch3 总共 3 个程序，就是为了给读者演示同一种问题的不同解决方法，在具体的使用中，读者还应该根据实际情况灵活应用。

13.4 进程的隐藏

进程隐藏技术用得最多的地方就是在病毒和木马中，因为这些不适合出现在阳光下的程序，越隐蔽生存率就越高。在当今 Windows 环境下，病毒和木马流传得越来越广泛，让读者适当了解这方面的技术可以在防治方面起到积极的作用，技术这种东西就是这样，大家都知道的“秘技”也就不再是“秘技”了，所以，大家都知道了进程隐藏是怎么一回事，进程隐藏起来也就不那么隐蔽了。

13.4.1 在Windows 9x 中隐藏进程

在 Windows 9x 系列操作系统中，可以通过 Kernel32.dll 中的一个未公开函数来完成隐藏功能，这个函数就是 RegisterServiceProcess，该函数的功能是将一个进程注册为系统服务进程，由于 Windows 的任务管理器并不列出系统服务进程，所以可以用它来隐藏进程，不过该函数在 Windows NT 系列中并不存在。

RegisterServiceProcess 函数的使用方法是:

invoke	RegisterServiceProcess, dwProcessID, dwFlag
--------	---

dwProcessID 指明目标进程的进程 ID，参数 dwFlag 指定是注册还是撤销，指定 TRUE 的话，进程被注册为系统服务进程，如果指定为 FALSE，则进程的属性恢复为普通进程属性。

Kernel32.lib 导入库中并没有这个函数的导入信息，如果要使用这个函数，程序需要自己装入库文件并使用 GetProcAddress 函数获取入口地址后使用（具体的方法请复习第 11 章）。所附光盘的 Chapter13\HideProcess9x 目录中的例子程序演示了该函数的使用方法。汇编源代码 HideProcess9x.asm 的内容如下：

[illegible]

由于可以确认 Kernel32.dll 库已经被装载到进程的地址空间中（GetProcAddress 等函数就包括在这个库中，因此这个库肯定已经被装入），所以例子中使用 GetModuleHandle 函数而不是使用 LoadLibrary 函数来获取库句柄，这样就可以省去一个 FreeLibrary 的调用。接下来，程序使用 GetProcAddress 函数获取 RegisterServiceProcess 函数的入口地址，如果获取成功，则使用 GetCurrentProcessId 函数获取当前进程的 ID 并将这个 ID 注册为系统服务进程。

13.4.2 Windows NT 中的远程线程

当然，如果不用进程也能运行程序的话，那是最好不过的办法了，但是不用进程是无法执行文件的。

在 Windows NT 中还有另一种办法，那就是使用远程线程，使用它可以在其他进程中创建一个线程，由于线程是被所属进程拥有的，所以任务管理器中列出来的还是所属进程的名称。

有两个函数可以用来实现上述功能：VirtualAllocEx 和 CreateRemoteThread。这两个函数都只能在 Windows NT 下使用。

VirtualAllocEx 函数的用法是:

491

```

, endif

```

在 10.1.5 节中已经介绍过虚拟内存管理函数 `VirtualAlloc`, `VirtualAllocEx` 函数就是这个函数的扩充, 相比之下, `VirtualAllocEx` 函数多了一个参数 `hProcess`, 其他参数定义和使用的方法都和 `VirtualAlloc` 函数相同, 读者可以回过头去查看这些参数的用法。新增的 `hProcess` 参数用来指定要申请内存的进程句柄, 如果 `hProcess` 指定的不是当前进程本身, 那么申请到的就是其他进程地址空间中的内存!

如果内存申请成功，函数返回一个指针，指向申请到的内存块，当然这个指针是针对目标进程的地址空间的。如果内存申请失败，函数返回 NULL。注意：如果需要在目标进程中使用 VirtualAllocEx 函数，那么必须对进程拥有 PROCESS_VM_OPERATION 权限。

CreateRemoteThread 函数用来在其他进程内创建一个线程，当然创建的线程也是运行于目标进程的地址空间内的，它和目标进程自己创建的线程并没有什么区别。函数的用法是：

invoke	CreateRemoteThread, hProcess, lpThreadAttributes, dwStackSize, \ lpStartAddress, lpParameter, dwCreationFlags, lpThreadId
--------	---

该函数是 `CreateThread` 函数的扩充，与 `CreateThread` 相比，`CreateRemoteThread` 函数多了一个 `hProcess` 参数，其他所有参数的定义和用法都与 `CreateThread` 的参数相同。`hProcess` 用来指定要创建线程的目标进程句柄。注意：`lpStartAddress` 指向的线程函数的地址是位于目标进程的地址空间内的。如果需要在目标进程中使用 `CreateRemoteThread` 函数，那么必须对进程拥有 `PROCESS_CREATE_THREAD` 权限。

使用 VirtualAllocEx 和 CreateRemoteThread 函数，再配合 WriteProcessMemory 函数，就能够让一段代码在其他进程中运行，由于远程线程是属于目标进程的，所以在任务管理器中不会产生新的进程，事实上，谁也不会发现列出的某个进程中会多了一个不属于它自己控制的线程。整个实现的过程归纳如下：

(1) 使用 VirtualAllocEx 函数在目标进程中申请一块内存，内存块的长度必须能够容纳线程使用的代码和数据，内存块的属性应该是 PAGE_EXECUTE_READWRITE，这样拷贝到内存块中的代码就可以被执行。

(2) 使用 `WriteProcessMemory` 函数将需要在远程线程中执行的代码(包括它使用的数据)拷贝到第(1)步申请到的内存块中。

(3) 使用 `CreateRemoteThread` 函数创建远程线程。

2. 远程线程存在的技术问题

实现远程线程的框架结构已经搭好了，但是在具体的实现中还有一些技术问题需要解决，归纳起来主要有两点：代码的重定位问题和函数的导入问题。

代码的重定位问题可以用下面的例子来说明，先来看一段简单的源代码：

```
dwVar          dd    ?
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Proc1          proc _dwParam
               local   @dwLocal
```

这段源代码中包括了调用子程序，存取全局变量、局部变量和参数的情况，经过编译链接以后再反汇编，就变成了下面的样子：

分析一下机器码就可以发现，存取全局变量的指令 `mov eax, dwVar` 中，全局变量的地址是包含在机器码中的（指令的机器码是 `A1FC0F4000`，第一个字节 `A1h` 是 `mov eax, xxx` 的机器码，后面的 `FC0F4000` 按照高字节在后的顺序读就是变量的地址 `00400FFCh`）；存取局部变量和参数的指令中并不包含绝对地址；`call` 指令中的地址数据也是相对的，所以，当这段机器码原封不动地从 `00401000h` 地址被搬到 `00801000h` 处的时候，就变成了下面的样子：

这时候, A1FC0F4000 机器码还是被解释为存取 00400FFCh 地址, 而实际的变量地址已经被搬到 00800FFCh 处了, 这就是说, 指令存取的是错误的地址, 所以这段指令要想正常执行, 就必须放在 00401000h 地址开始的地方, 如果想搬到别的地方去执行, 就必须对访问全局变量的指令进行修正, 这就是重定位的问题

493

对于高级语言来说，重定位问题是个致命的问题，是根本不可能解决的，因为高级语言无法在机器码级别上进行细微的操作，所以，即使在相对比较低级的 C 语言中也无法将一段代码拷贝到远程线程中去执行，大部分的教科书和资料在介绍远程线程的时候，都采用了变通的方法，就是将 DLL 嵌入到目标进程中去执行。

如 Jeffrey Richer 的《Windows 高级编程指南》中就介绍了使用远程线程将 DLL 注入目标进程的方法，其实现步骤是将需要远程执行的代码写到一个 DLL 中，然后在目标进程中申请一块内存并将 DLL 文件名写入，最后将目标进程地址空间中的 LoadLibrary 函数当做线程函数来执行，输入的参数就是前面的 DLL 文件名，这样 LoadLibrary 函数执行到 ret 的时候，远程线程结束，但是 DLL 也被装入了目标进程中，在这个 DLL 的入口函数中创建一个新的线程，就可以执行指定的代码了。这种方法以系统装入 DLL 时会自动重定位的方法回避了重定位问题。

在所附光盘的 Chapter13\RemoteThreadDll 中的例子演示了这种方法的汇编版本，程序将一个 DLL 文件插入到文件管理器 Explorer.exe 中运行，有兴趣的读者可以自己分析一下。

虽然 DLL 文件在目标进程中运行的时候，任务管理器中不会列出 DLL 文件名，看到的只是目标进程的文件名，但是有一些工具可以查看一个进程究竟装入了哪些 DLL 文件，通过这些工具还是可以发现进程中的可疑 DLL。

要彻底解决这个问题，就必须脱离 DLL 文件，让远程运行的代码只存在于内存中，这样就不会有任何的蛛丝马迹显示有某个文件被非法装入，这个问题的关键也就是这个重定位问题。但现在 Win32 汇编程序员可以很骄傲地说“我可以实现它”，因为自定位的代码正是汇编语言的拿手好戏，在快成为历史的 DOS 病毒中，10 个病毒中就有 9 个用到了自定位技术，这些技术完全可以用在这个地方。

自定位技术其实很简单，观察下面这段代码：

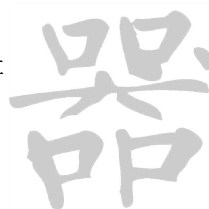
dwVar	dd	?
	call	@F
	@@:	
	pop	ebx
	sub	ebx, offset @B
	mov	eax, [ebx + offset dwVar]

翻译成机器码就是：

:00401000	00000000	BYTE 4 DUP(0)
:00401004	E800000000	call 00401009
:00401009	5B	pop ebx
:0040100A	81EB09104000	sub ebx, 00401009
:00401010	8B8300104000	mov eax, dword ptr [ebx+00401000]

这段代码不存在重定位的问题，分析如下。

call 指令会将返回地址压入到堆栈中，当整段代码在没有移动的情况下执行的时候，call @F 指令执行后堆栈中的返回地址就是 @@ 标号的地址 00401009h，下一句 pop 指令将返回地址弹出到 ebx 中，再接下来 ebx 减去 00401009h，现在 ebx 等于 0，所以 mov eax, [ebx + offset dwVar] 指令就等于 mov eax, dwVar 指令。



当整段代码被移动到其他地方时（假设被移到 00801000 处执行），@@标号现在对应的地址是 00801009h，变量 dwVar 的地址对应 00801000h，当 call 指令执行后，压入到堆栈的地址是 00801009h，pop 到 ebx 中的就是这个数值，经过 sub ebx, 00401009 指令以后，ebx 等于 00400000h，现在 mov eax, dword ptr [ebx+00401000] 指令就相当于 mov eax, [00801000]，而 00801000 这个地址刚好等于 dwVar 现在所处的位置，所以，虽然代码被移动了位置，mov eax, dwVar 指令还是访问了正确的地方。

call/pop/sub 这 3 个指令组合的用途就是计算出代码当前的位置和设计时位置的偏移值之差，只要用这个差值去修正包含绝对地址的指令，如访问全局变量的指令，就能够保证修正后的地址是正确的，这就解决了重定位的问题。

另一个问题就是函数的导入问题，由于 Win32 编程不可避免地要用到 API 函数，而 API 函数又存在于 DLL 中，当远程代码要用到一个 API 函数时，就必须保证目标进程中已经装入了相应的 DLL，还必须知道 API 函数的地址，否则对函数的调用就无从谈起。

所以在设计远程代码的时候，不能直接使用 API 函数，因为函数的地址在不同的进程中会随着 DLL 装入位置的不同而不同，如果在代码中直接调用 API 函数，那么系统会按照当前进程的 DLL 装入位置填入函数地址，但这个地址搬到远程线程中可能是错误的。

要在远程代码中使用 API 函数，就必须手动完成本来由系统完成的工作，那就是自己装入每个要使用的 DLL，并使用 GetProcAddress 函数获取全部要使用的 API 函数的入口地址。由于这个过程要用到 DLL 文件的名称和函数名称，这些字符串必须放在全局变量中，这就又遇到了重定位的问题（所以在高级语言中实现函数的手动导入也是个很大的麻烦），当然现在这个问题是很容易解决的。

3. 远程线程的具体实现

好了，经过这么长时间的纸上谈兵，现在动真格的（本节中的所有源程序都可以在所附光盘的 Chapter13\RemoteThread 目录中找到），还记得第 4 章中的窗口例子吗？我们的目标就是将这个创建窗口的程序整个搬到 Windows 的文件管理器 Explorer.exe 进程中去执行。选定文件管理器开刀的原因是，它是 Windows 的“常任理事”，这个进程任何时刻都在运行，所以不必担心找不到它。

读者可能会问，难道连包含消息循环、窗口过程的代码都可以放到远程线程中去执行吗？当然可以，因为第 12 章中已经介绍过，每个线程的消息队列是独立的，远程线程的消息循环并不会和 Explorer.exe 程序原来的消息循环互相混淆。

工作的第一步就是设计远程线程使用的代码，代码中必须包括一个线程函数用做远程线程执行的入口，在线程函数中必须完成所有所需 DLL 的装入工作和 API 函数地址的获取工作，然后调用创建窗口的主程序；第二步就是修改所有访问全局变量的代码，以解决重定位问题。

完工后的远程代码存放在 RemoteCode.asm 文件中：

```
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
REMOTE_CODE_START equ this byte
```

```

_lpLoadLibrary          dd      ?      ;导入函数地址表
_lpGetProcAddress      dd      ?
_lpGetModuleHandle      dd      ?

_lpDestroyWindow        dd      ?
_lpPostQuitMessage      dd      ?
_lpDefWindowProc        dd      ?
_lpLoadCursor           dd      ?
_lpRegisterClassEx      dd      ?
_lpCreateWindowEx       dd      ?
_lpShowWindow           dd      ?
_lpUpdateWindow         dd      ?
_lpGetMessage           dd      ?
_lpTranslateMessage     dd      ?
_lpDispatchMessage      dd      ?

_hInstance              dd      ?
_hWinMain               dd      ?
_szClassName            db      'RemoteClass',0
_szCaptionMain          db      'RemoteWindow',0
_szDestroyWindow        db      'DestroyWindow',0
_szPostQuitMessage      db      'PostQuitMessage',0
_szDefWindowProc        db      'DefWindowProcA',0
_szLoadCursor           db      'LoadCursorA',0
_szRegisterClassEx      db      'RegisterClassExA',0
_szCreateWindowEx       db      'CreateWindowExA',0
_szShowWindow           db      'ShowWindow',0
_szUpdateWindow         db      'UpdateWindow',0
_szGetMessage           db      'GetMessageA',0
_szTranslateMessage     db      'TranslateMessage',0
_szDispatchMessage      db      'DispatchMessageA',0,0
_szDllUser              db      'User32.dll',0
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_RemoteThread           proc      uses ebx edi esi lParam
                        local     @hModule

                        call      @F
@@:
                        pop       ebx
                        sub       ebx,offset @B
;*****
                        _invoke   [ebx + _lpGetModuleHandle],NULL
                        mov       [ebx + _hInstance],eax
                        lea       eax,[ebx + offset _szDllUser]
                        _invoke   [ebx + _lpGetModuleHandle],eax
                        mov       @hModule,eax
                        lea       esi,[ebx + offset _szDestroyWindow]
                        lea       edi,[ebx + offset _lpDestroyWindow]
                        .while    TRUE
                        _invoke   [ebx + _lpGetProcAddress],@hModule,esi
                        mov       [edi],eax
                        add       edi,4
                        @@:
                        lodsb

```


[illegible]

在上面的代码中，所有对 API 函数的调用被换成了对函数入口地址的调用，因为入口地址被存放在全局变量中，所以要用 `call [ebx + XXXX]` 的格式调用以解决重定位问题，但是这样的话就无法使用 `invoke` 伪指令了。

因为用一大堆的 push 指令来压入参数太麻烦，笔者写了一个宏来自动压入参数，宏的名称定为 invoke，宏定义存放在 Macro.inc 文件中。文件的内容是：

[illegible]

远程代码中的线程函数是 `_RemoteThread`，在这里程序首先获取要用到的 API 函数的地址，所有 API 函数的名称被放到了一系列相连的字符串中，最后以一个附加的 0 结束，这样可以很方便地通过循环来处理它们。

要获取函数地址必须使用 LoadLibrary, GetProcAddress 和 GetModuleHandle 函数, 但这些函数地址又从哪里得到呢 (这就好像一个 “先有鸡, 还是先有蛋” 的问题), 幸亏这些函数都存在于 Kernel32 库中, Kernel32.dll 库文件和 User32.dll, Gdi32.dll 一样, 都是最常用的库, 在不同的进程中, 系统基本上会将它装入到同样的地址中, 所以对于它们来说, 在本地进程中获取的地址可以用在远程线程中。

如果从绝对保险的角度考虑，必须要在目标进程中得到这些函数地址的话，也可以使用第17章第6节中动态获取API入口地址的方法——搜寻目标进程的整个地址空间来得到函数的入口地址，不过那样会使程序复杂不少，为了让例子简单起见，例子代码中暂时使用上一段假设中的结论，也就是用本地这3个函数的地址代替远程函数的地址。

完成获取 API 函数地址的工作后，就可以调用 `_WinMain` 函数来创建窗口了，注意在 `_WinMain` 函数的后面不能使用 `ExitProcess` 函数来结束进程，这样会将整个 `Explorer.exe` 结束掉，必须使用 `ret` 指令来结束线程。

_WinMain 函数和其他的相关代码改编自第 4 章中的 FirstWindow.asm 程序,只不过是程序中所有涉及全局变量的指令全部改成了以 ebx 为基址的指令而已。另外,在所有的子程序的开始处,都加上了 call/pop/sub 这 3 句用来计算偏移差的指令。

完成远程线程的代码后，现在来看如何将这段代码装载到目标进程中，装载代码存放在 RemoteThread.asm 中：

```

        .386
        .model flat, stdcall
        option casemap :none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; Include 文件定义
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include      windows.inc
include      user32.inc
includelib   user32.lib
include      kernel32.inc
includelib   kernel32.lib
include      Macro.inc
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .data      ?
lpLoadLibrary      dd      ?
lpGetProcAddress    dd      ?
lpGetModuleHandle   dd      ?
dwProcessID         dd      ?
dwThreadID          dd      ?
hProcess            dd      ?
lpRemoteCode        dd      ?
dwTemp              dd      ?

        .const
szErrOpendb         '无法打开远程线程!', 0
szDesktopClass       db      'Progman', 0
szDesktopWindow      db      'Program Manager', 0
szDllKernel          db      'Kernel32.dll', 0
szLoadLibrary        db      'LoadLibraryA', 0
szGetProcAddress     db      'GetProcAddress', 0
szGetModuleHandle    db      'GetModuleHandleA', 0
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .code
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include      RemoteCode.asm
start:
invoke       GetModuleHandle, addr szDllKernel
mov          ebx, eax

```

哭

接下来就是打开 Explorer.exe 进程的操作，程序通过 `GetWindowThreadProcessId` 和 `OpenProcess` 函数来完成，函数中使用的窗口句柄是桌面的窗口句柄，因为桌面就是由文件管理器进程创建的，桌面的窗口类是 “Progman”，窗口名称是 “Program Manager”，使用 `FindWindow` 函数就可以很方便地找到它。在打开进程的时候必须包括对应的权限，`PROCESS_CREATE_THREAD` 权限将允许创建远程线程，`PROCESS_VM_OPERATION` 权限将允许在目标进程中分配内存并将远程代码写到里面。

程序使用 VirtualAllocEx 函数在目标进程中分配内存，在分配内存的时候，内存属性必须指定为 PAGE_EXECUTE_READWRITE，这样分配到的内存可以有执行和读写的权限，分配方式必须指定为 MEM_COMMIT，这样内存才会被提交到物理内存中去。

在分配到内存以后，程序使用 WriteProcessMemory 将远程代码写入，然后再一次将 LoadLibrary, GetProcAddress 和 GetModuleHandle 函数的地址写入到远程代码的数据段中。并不将这 3 个函数的地址存放到远程代码中一次性写入的原因在于：远程代码（包括远程代码使用的数据）是定义在本地的代码段中的，而本地的代码段是只读的，我们无法在本地对它们进行写入初始化数据的操作，所以只好采用远程写入的方式。

最后，用 CreateRemoteThread 函数创建远程线程后就万事大吉了。编译链接以后运行可执行文件可看到，窗口正常出现了，一眼看上去，这个窗口和别的窗口没有任何不同！但是在任务管理器中却没有多出任何新的进程。

假如远程线程不是这样“招摇过市”地创建了一个窗口，而是在后台偷偷地运行的话，大家能不能从各种蛛丝马迹来发现它的存在呢？反正笔者是找不到它的，因为它仅存在于目标进程的内存中，并不对应任何磁盘文件，当远程线程被执行的时候，惟一可以发现的就是 Explorer.exe 进程中的活动线程多了一个，使用的内存多了一点而已，但是活动线程用工具软件查看也只能看到一个线程 ID，又怎么知道这个线程不是 Explorer.exe 进程自己的呢？



以上代码用在不合适地方可能产生危害，笔者公开这段代码，其目的就是希望能对有害代码的防治起到积极的作用，请读者负责任地使用这段代码。

第 14 章

异常处理

14.1 异常处理的用途

对程序执行中的异常（Exception）大家都不陌生，先来回顾一下 DOS 操作系统对异常的处理方法。在 DOS 操作系统中，当操作系统在运行中发生“异常”时，会去调用 INT 24h 中断，系统默认的 INT 24h 中断处理程序会将出错代码翻译成文本信息显示在屏幕上，然后让用户选择“Ignore”，“Retry”，“Fail”或者“Abort”，并根据用户的输入结果来选择不同的操作——忽略异常、重试产生异常的操作或者强行终止程序的执行。

系统默认的异常处理方法有时候不是很合适，比如，对于在图形方式下运行的 DOS 程序来说，系统显示的出错信息会破坏屏幕的美观；另外，程序可能希望不要向用户提供“Abort”选项来保证程序不会因为一些微不足道的错误而被终止。为了处理这些情况，程序可以用自定义的异常处理程序来替换系统默认的处理程序，而这是很容易实现的——由于 DOS 系统在检测到异常的时候仅仅去调用 24h 号中断并根据中断的返回值决定下一步的处理方式，只要应用程序截获 INT 24h 中断，就可以在自己提供的中断服务程序中按照自己的意图决定系统处理异常的走向。

Windows 操作系统对异常的处理流程相对比较复杂，与 DOS 操作系统相比，最大的区别在于 DOS 的异常处理是被动的，一般仅用来处理操作系统内部的异常，对于其他层次的异常是无法处理的，比如，使用 INT 21h 去读盘的时候发生错误会激发 INT 24h 中断，但在 BIOS 服务程序级别用 INT 13h 去读盘时发生错误就不会激发 INT 24h 中断，对应用程序胡作非为引发的异常更是束手无策；而 Windows 的异常处理机制是依靠 80x86 处理器的保护机制来主动捕获异常，所以 Win32 下异常处理程序的用途不仅仅局限于防止程序被 Windows 野蛮地终止，合理利用它们可以让有些功能的实现方式变得更加简单，一般来说，可以在下面这些情况下使用异常处理程序。

- 用来处理非致命的错误

程序执行中发生某些异常时只需要终止发生异常的模块（或子程序），并没有必要终止整个程序的运行，这时可以在异常处理程序中指定让程序转移到一个“安全”的地方去执行，并在这

里完成资源释放、删除临时文件、显示错误提示等扫尾工作后从出错模块返回。

● 处理“计划内”的异常

程序中的有些功能本来就是设计在异常处理模块中实现的。Windows 系统中虚拟内存的实现就是一个绝好的例子（如图 1.5 所示），第 10 章中介绍的内存映射文件也是以同样的方法实现的。在这些情况下，“异常”是作为一个触发条件使用的。

另外，在 Windows API 中使用异常处理程序进行参数的合法性检测也是很常见的。一般来说，大部分子程序都需要对输入的参数进行合法性检测，特别是对于指针类型的参数，但是当参数涉及的数据结构太复杂的时候，合法性检测会大大降低程序的效率，这时可以假定参数全部合法并尝试直接使用这些参数，如果异常处理程序没有捕获到错误，那么表示参数是合法的，这样要比在每个步骤中检测参数（或操作结果）的合法性要简洁得多。

● 处理致命错误

虽然捕获到致命错误的时候终止程序是最好的选择，但是程序在退出之前，可以在异常处理程序中进行释放资源、删除临时文件等操作，甚至可以详细记录产生异常的指令位置和环境，以便用来分析产生异常的原因。

显然，Windows 中的用户自定义的异常处理函数不会再以 INT 24h 的方式被调用，读者也可以猜到它必定会以“回调函数”的方式来实现，但是如何写回调函数，回调函数的参数是什么，在什么地方定义回调函数呢？接下来将介绍这些内容。

14.2 使用筛选器处理异常

Windows 下的异常处理可以有两种方式：筛选器异常处理和 SEH 异常处理。

筛选器异常处理的方式是由程序指定一个异常处理回调函数（在下面将统一简称为“回调函数”），当发生异常的时候，系统将调用这个回调函数，并根据回调函数的返回值决定如何进行下一步操作，这种方法和 DOS 系统中使用 INT 24h 中断来处理异常的方法是很像的。

在进程范围内，筛选器异常处理回调函数是惟一的，设置了一个新的回调函数后，原来的就失效了。

14.2.1 注册回调函数

可以使用 SetUnhandledExceptionFilter 函数来设置一个筛选器异常处理回调函数，准确地讲，这个回调函数不是替换了系统默认的异常处理程序，而是在它前面进行了一些预处理，操作的结果还是会被送到系统默认的异常处理程序中去，这个过程就相当于对异常进行一次“筛选”，这正是函数名中“Filter”一词的含义。

SetUnhandledExceptionFilter 函数的使用方法是：

invoke	SetUnhandledExceptionFilter, offset Handler
mov	lpPrevHandler, eax

器

程序的入口处使用 `SetUnhandledExceptionFilter` 函数将 `_Handler` 子程序指定为异常处理回调函数，函数返回的原回调函数地址被保存到 `lpOldHandler` 变量中（当然，在这个例子中，这个值肯定为 0，程序中进行这一步操作是为了演示保存和恢复的方法），在程序退出之前会再次使用 `SetUnhandledExceptionFilter` 函数将这个地址设置回去。

14.2.2 异常处理回调函数

_Handler	proc	lpExceptionInfo
----------	------	-----------------

```

EXCEPTION_POINTERS STRUCT
    pExceptionRecord DWORD    ?
    ContextRecord     DWORD    ?
EXCEPTION_POINTERS ENDS

```

506

结构中记录了异常产生时刻的运行环境。这两个结构的定义在 13.3.3 节中介绍调试 API 的时候介绍过。

1. 获取产生异常的原因

重新来看一下 EXCEPTION_RECORD 结构的定义：

```
EXCEPTION_RECORD STRUCT
    ExceptionCode      DWORD      ?      ;异常事件码
    ExceptionFlags      DWORD      ?      ;标志
    pExceptionRecord    DWORD      ?      ;下一个 EXCEPTION_RECORD 结构地址
    ExceptionAddress     DWORD      ?
    NumberParameters    DWORD      ?
    ExceptionInformation DWORD      EXCEPTION_MAXIMUM_PARAMETERS dup(?)
EXCEPTION_RECORD ENDS
```

结构中的 ExceptionCode 字段定义了产生异常的原因，这些原因已经被预定义为一系列以 EXCEPTION_ 开头或者以 STATUS_ 开头的常量。表 14.1 中列出了一些最常用的异常原因。除了表中列出的原因之外，系统中还定义了许多其他异常原因代码，由于 MASM32 SDK 软件包中所带的 Windows.inc 文件中的定义值也不是很详尽，为此笔者整理了一份详细的异常原因代码文档来供读者参考，文档放在本书所附光盘的 Chapter14\Exception.inc 文件中，文档中定义的代码都是以 STATUS_ 开头的，但它们在数值上与以 EXCEPTION_ 开头的代码是一样的。

由于例子中的 mov dword ptr [eax], 0 指令去写一个没有写权限的地址，所以会引发一个 EXCEPTION_ACCESS_VIOLATION 异常，从例子程序运行后显示的消息中可以验证这点，读者也可以尝试将这条指令修改为各式各样的错误指令，看看它们引发的异常对应哪个异常原因代码。

表 14.1 异常原因代码的列表

异常原因	对应值	说明
EXCEPTION_ACCESS_VIOLATION	0C0000005h	尝试读写没有可读写属性的地址
EXCEPTION_BREAKPOINT	080000003h	断点异常（遇到 INT 3 指令）
EXCEPTION_ILLEGAL_INSTRUCTION	0C000001Dh	遇到无效指令
EXCEPTION_IN_PAGE_ERROR	0C0000006h	存取不存在的内存页面
EXCEPTION_INT_DIVIDE_BY_ZERO	0C0000094h	除零错误
EXCEPTION_SINGLE_STEP	080000004h	单步中断
EXCEPTION_STACK_OVERFLOW	0C00000FDh	堆栈溢出
EXCEPTION_UNWIND	0C0000027h	展开操作

异常原因代码的含义是按照数据位划分的，其规则如表 14.2 所示。

表 14.2 异常原因代码各数据位的含义

数据位	含义	取值说明
位 31~30	严重性系数	00=成功，01=信息，10=警告，11=错误

位 29	定义者	0=Microsoft 定义，1=客户应用程序定义
位 28	保留位	必须为 0
位 27~16	设备代码	表示引发异常的位置
位 15~0	异常代码	用来分辨产生异常的原因

其中的位 27~16 定义了设备代码，用来表示异常代码发生在哪个特定的设备中，当前已经定义的设备代码如表 14.3 所示。

表 14.3 异常原因代码中的设备代码定义

设备代码	取值	设备代码	取值
FACILITY_NULL	0	FACILITY_CONTROL	10
FACILITY_RPC	1	FACILITY_CERT	11
FACILITY_DISPATCH	2	FACILITY_INTERNET	12
FACILITY_STORAGE	3	FACILITY_MEDIASERVER	13
FACILITY_ITF	4	FACILITY_MSMQ	14
FACILITY_WIN32	7	FACILITY_SETUPAPI	15
FACILITY_WINDOWS	8	FACILITY_SCARD	16
FACILITY_SECURITY	9	FACILITY_COMPLUS	17

举例来讲，由于断点异常和单步中断并不属于程序错误，所以这两种异常代码中的严重性系数 10，表示属于警告信息而非错误，但是遇到内存越权访问、除零错误等时候，这两位的值就是 11 了，表示这个错误让线程无法继续执行。

EXCEPTION_RECORD 结构中的 ExceptionCode 字段定义了异常标志，它由一系列的数据位构成，定义如下：

- 位 0——代表发生的异常是否允许被恢复执行。当位 0 被复位的时候，表示回调函数在对异常进行处理后可以指定让程序继续运行，置位的时候表示这个异常是不可恢复的，这时程序最好进行退出前的扫尾工作并选择终止程序，如果这时非要指定让程序继续执行的话，Windows 会再次以 EXCEPTION_NONCONTINUABLE_EXCEPTION 异常代码调用回调函数。为了程序的可读性，可以通过两个预定义值 EXCEPTION_CONTINUABLE（定义为 0）和 EXCEPTION_NONCONTINUABLE（定义为 1）来测试这个标志位。
- 位 1——EXCEPTION_UNWINDING 标志。表示回调函数被调用的原因是进行展开操作（详见 14.3.4 节）。
- 位 2——EXCEPTION_UNWINDING_FOR_EXIT 标志。表示回调函数被调用的原因是进行最终退出前的展开操作。

当处理异常的代码设计得不完善而在运行中引发新的异常时，回调函数会被嵌套调用。在这种情况下，EXCEPTION_RECORD 结构中的 pExceptionRecord 字段会指向下一个 EXCEPTION_RECORD 结构，这条 EXCEPTION_RECORD 结构链定义了嵌套发生的多个异常的情况；如果没有嵌套的异常，pExceptionRecord 的值为 NULL。

结构中的 ExceptionAddress 字段定义了引发异常的指令的地址。

有了这些信息后，回调函数就可以根据类型来对不同的异常进行不同的处理，比如，对那些“计划内”的异常执行功能性的代码，而发生其他非致命异常的时候转到“安全”位置去执行。

2. 修正错误

在筛选器异常处理回调函数被系统调用的时候，参数中指定的 `EXCEPTION_POINTERS` 结构中的 `ContextRecord` 字段指向一个 `CONTEXT` 结构，这个结构中保存了异常发生时刻的运行环境，也就是所有寄存器的值。程序也可以通过这个结构中的 `regEip` 字段来得知异常发生的位置。在例子中，程序用一个对话框显示出异常代码、异常标志和 `CONTEXT` 结构中的 `EIP` 值，对比一下就可以发现，显示的 `EIP` 值正是那句产生异常的 `mov [eax], 0` 指令的地址。

在第 12 章介绍多线程时，已经介绍过操作系统为每个线程保存单独的寄存器环境和单独的堆栈，那么当异常发生的时候，`CONTEXT` 结构指出的环境会对应哪个线程的环境呢？其实答案很简单：Windows 将会在产生异常的线程中运行回调函数，`CONTEXT` 结构对应的是出错线程的环境，回调函数使用的堆栈也是这个线程的堆栈，这很容易理解，因为只有这样的安排下，回调函数才可能去修复出错线程中的错误。

修正错误的操作反映在对这个 `CONTEXT` 结构的修改上，当回调函数修改了结构中的值并返回后，系统会将线程的运行环境设置为新的值，所以要修正某个寄存器中的错误取值，只要修改这个 `CONTEXT` 结构就可以了。在例子中，异常处理程序将 `regEip` 字段的值修改为 `_SafePlace` 标号的地址，这样线程恢复运行时是从 `_SafePlace` 标号的地方开始执行的。当然，这样处理后，在错误指令和 `_SafePlace` 标号之间的其他指令就不会被执行了。

3. 回调函数的返回值

回调函数返回后，Windows 执行默认的异常处理程序，这个程序会根据回调函数的返回值决定如何进行下一步动作。

回调函数的返回值可以有 3 种取值：`EXCEPTION_EXECUTE_HANDLER`（定义为 1）、`EXCEPTION_CONTINUE_SEARCH`（定义为 0）和 `EXCEPTION_CONTINUE_EXECUTION`（定义为 -1）。

当返回值是 `EXCEPTION_EXECUTE_HANDLER` 时，进程将被终止，但是在终止之前系统不会显示出错误提示对话框；当返回值是 `EXCEPTION_CONTINUE_SEARCH` 时，系统同样将终止程序的执行，但是在终止前会首先显示出错误提示对话框。使用这两种返回值的时候，异常处理程序完成的工作一般是退出前的扫尾工作。

而返回值是 `EXCEPTION_CONTINUE_EXECUTION` 时，系统将 `CONTEXT` 设置回去并继续执行程序，例子程序中就是这样使用的。



当异常标志中包含 `EXCEPTION_NONCONTINUABLE` 标志位时，不应该使用 `EXCEPTION_CONTINUE_EXECUTION` 作为返回值，这样只会引发一个新的异常。（例子程序中为了简化代码，没有判断并处理异常标志为 `EXCEPTION_NONCONTINUABLE` 的情况）

14.3 使用 SEH 处理异常

使用筛选器异常处理程序是最简单的处理异常的方法，但在使用中也不存在一些不便之处，最明显的就是不便于模块的封装：由于筛选器异常处理程序是全局性的，无法为一个线程或一个子程序单独设置一个异常处理回调函数，这样就无法将私有的异常处理代码封装进某个模块中。

Windows 系统中还提供了另一种在每个线程之间独立的异常处理方法——结构化异常处理（Structured Exception Handling, SEH），SEH 是 Win32 系统中为数不多的应用广泛却又未被公开的特征之一。

SEH 和筛选器异常处理之间有一些共同点：首先是两者的异常处理程序都是以回调函数的方式提供的；另外，系统都会根据回调函数的返回值选择不同的操作。

但是它们之间也存在很多的不同点：

- SEH 是基于线程的，使用 SEH 可以为每个线程设置不同的异常处理程序，而且可以为每个线程设置多个异常处理程序。
- 两者的回调函数的调用类型、参数定义和返回值的定义都是不同的。
- 由于 SEH 使用了与硬件平台相关的数据指针，所以在不同硬件平台中使用 SEH 的方法会有所不同（也许这正是 SEH 未被 Microsoft 公开的原因）。

接下来首先看一个使用 SEH 处理异常的例子，这个例子与前面的例子很相似，都是在回调函数中显示异常代码和发生异常的位置，并将程序修正到 _SafePlace 标号去执行。例子的源代码可以在本书所附光盘的 Chapter14\SEH01 目录中找到，其中的 SEH.asm 的内容如下：

```
.386
.model flat, stdcall
option casemap:none
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include      windows.inc
include      user32.inc
includelib   user32.lib
include      kernel32.inc
includelib   kernel32.lib
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        .const
szMsg    db      '异常发生位置: %08X, 异常代码: %08X, 标志: %08X', 0
szSafe   db      '回到了安全的地方!', 0
szCaption db      'SEH 例子', 0

        .code
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; SEH Handler 异常处理程序
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_Handler proc    C _lpExceptionRecord, _lpSEH, \
```


14.3.1 注册回调函数

在例子程序中，SEH 异常处理回调函数的设置由下面 3 条指令完成：

```
push offset _Handler
push fs:[0]
mov     fs:[0], esp
```

为什么这 3 句简单的指令就可以完成设置工作，为什么又要使用 fs 段选择器呢？这要从线程信息块（Thread Information Block/TIB）说起。

Win32 为每个线程定义了一个线程信息块，其中保存了线程的一些属性数据，线程信息块的格式被定义为 NT_TIB 结构：

```
NT_TIB STRUCT
    ExceptionList dd      ?      ;SEH 链入口
    StackBase     dd      ?      ;堆栈基址
    StackLimit     dd      ?      ;堆栈大小
    SubSystemTib  dd      ?
    FiberData      dd      ?
    ArbitraryUserPointer dd ?
    Self           dd      ?      ;本 NT_TIB 结构自身的线性地址
NT_TIB ENDS
```

NT_TIB 结构的第一个字段 ExceptionList 指向一个 EXCEPTION_REGISTRATION 结构，SEH 异常处理回调函数的入口地址就是由 EXCEPTION_REGISTRATION 结构指定的，这个结构的定义如下：

```
EXCEPTION_REGISTRATION STRUCT
    prev dd      ?      ;前一个 EXCEPTION_REGISTRATION 结构的地址
    handler dd      ?    ;异常处理回调函数地址
EXCEPTION_REGISTRATION ENDS
```

当异常发生时，系统从 TIB 中取出 ExceptionList 字段，然后从 ExceptionList 字段指定的 EXCEPTION_REGISTRATION 结构中取出 handler 字段，并根据其中的地址去调用回调函数，整个过程如图 14.1 所示，所以用户程序想自定义一个异常处理程序的话，只要构建一个新的含有回调函数地址的 EXCEPTION_REGISTRATION 结构，然后修改 TIB 中的 ExceptionList 字段，将其指向这个结构就可以注册一个 SEH 异常处理回调函数了。

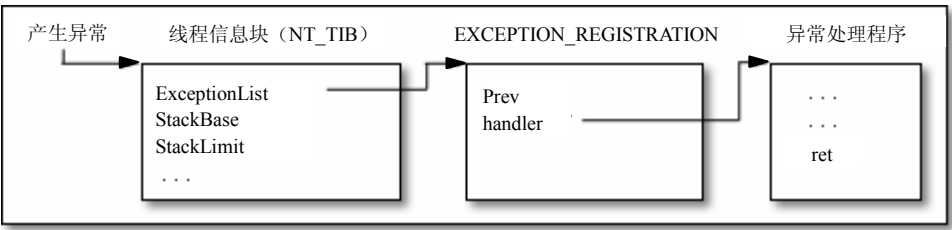


图 14.1 SEH 异常处理程序入口地址的定义

现在还剩下一个关键的问题：到哪里找 TIB 呢？答案是：TIB 永远放在 fs 段选择器指定的

数据段的 0 偏移处，所以，fs:[0]的地方就是 TIB 结构的 ExceptionList 字段，这个答案对于 Windows 9x 系统和 Windows NT 系统都是有效的。由于一个进程中的不同线程可以有不同的环境，所以，在不同线程中 fs 段选择器可以使用不同的值，这种特征使每个线程都可以设置不同的回调函数。



也正是因为使用了 fs 段选择器，所以使 SEH 变得与硬件平台相关，试想一下，Power PC 或者 Alpha 平台上有 fs 段选择器吗？

好了，现在回过头来看看用于设置回调函数的 3 条指令，第一条 push offset _Handler 指令将回调函数的地址推入堆栈；第二条 push fs:[0] 指令则将原先使用的 EXCEPTION_REGISTRATION 结构地址推入堆栈，现在堆栈指针 esp 指向的地方刚好是一个新的 EXCEPTION_REGISTRATION 结构——[esp]等于原结构地址，也就是 prev 字段，而[esp+4]等于回调函数地址，也就是 handler 字段；当第三条指令 mov fs:[0], esp 将 esp 的值放入 fs:[0] 后，设置工作就完成了。

当不再需要这个回调函数的时候，只要将 fs:[0] 的值恢复为原来的 EXCEPTION_REGISTRATION 结构地址就可以了，这个地址已经被保存在 prev 字段中，例子程序中使用下面的恢复代码：

```
pop fs:[0]
pop eax
```

第一条指令从堆栈中的 prev 字段中弹出原来的 fs:[0] 值；第二条指令 pop eax 仅仅是为了让堆栈平衡，弹出到 eax 中的值没有实际用途。执行了这两条指令后，堆栈中废弃的 EXCEPTION_REGISTRATION 结构也被释放掉了。

例子程序在堆栈中构造 EXCEPTION_REGISTRATION 结构，而不是将结构放在全局的数据段中，这实际上就是局部变量的使用方法，这使构建异常处理程序使用的数据结构存放在一个子程序的私有空间中，更有利于程序结构的模块化。实际上，所有的高级语言在使用 SEH 时都将数据结构建立在堆栈中。



由于 MASM 编译器默认将 fs 段寄存器定义为 error，所以程序在使用 fs 之前要用 assume fs:nothing 伪指令来启用 fs 寄存器，否则编译的时候会产生下面的错误：

```
error A2108: use of register assumed to ERROR
```

14.3.2 异常处理回调函数

1. 回调函数的参数

SEH 异常处理回调函数的参数定义与筛选器回调函数的参数定义有所不同，其定义如下，注意：回调函数的调用类型不是 std call 的，而是 C 格式的：

```
_Handler proc C _lpExceptionRecord, _lpSEH, _lpContext, _lpDispatcherContext
```

在这个回调函数中，前面的 3 个参数是要用到的。其中的 `_lpExceptionRecord` 参数指向一个 `EXCEPTION_RECORD` 结构；`_lpContext` 参数指向一个 `CONTEXT` 结构，这两个结构提供的数据就相当于 14.2.2 节中筛选器回调函数从参数中得到的数据，可以用同样的方法来使用它们；`_lpSEH` 参数指向注册回调函数时使用的 `EXCEPTION_REGISTRATION` 结构的地址，在例子程序中，它的值就是我们在堆栈中构造的这个结构的地址，这个参数看上去似乎没有什么用处，例子程序中也确实没有用到它，但是如果希望异常处理程序能够被封装在子程序里面的话，这个参数就是不可缺少的，因为使用它可以避免使用全局变量在模块和回调函数之间传递数据，在接下来的内容中读者会了解到如何做到这一点。

本章的前两个例子为了简便起见，演示的都是在主程序中执行异常指令的情况，现在来考虑产生异常的指令发生在子程序中的情况，演示的指令序列如下所示：

<code>_Test</code>	<code>proc</code>
	<code>pushad</code>
	<code>xor ebp, ebp</code>
	<code>xor eax, eax</code>
	<code>mov dword ptr [eax], 0 ; 异常指令</code>
	<code>...</code>
<code>_SafePlace:</code>	<code>popad</code>
	<code>ret</code>
<code>_Test</code>	<code>endp</code>

这段代码首先将所有的寄存器入栈，然后使用了 `ebp` 寄存器和 `eax` 寄存器，最后结束的时候使用 `popad` 从堆栈中恢复所有的寄存器。如果不产生异常的话，那么指令执行完毕以后，`ebp` 的值是正常的，堆栈也是平衡的；但是如果中间的某条指令产生异常的话（就像上面代码中 `mov dword ptr [eax], 0` 这个位置的指令），回调函数必须在将程序修正到“安全”位置去执行的同时恢复 `esp` 和 `ebp` 的值，否则，由于 `ebp` 和 `esp` 值被破坏，程序还是会崩溃。

要将回调函数写得比较“强壮”的话，就必须考虑到这种可能性。最安全的办法就是预先保存一些关键寄存器的值，如果发生异常，回调函数就可以根据保存的数据恢复这些关键寄存器。这些关键寄存器值可以预先保存在全局变量中，但为了程序的模块化，一般推荐使用堆栈来动态传递数据。

如何使用 `_lpSEH` 参数实现用堆栈传输数据呢？大家可以注意到，`_lpSEH` 参数指向的数据就是我们自己定义的 `EXCEPTION_REGISTRATION` 结构，这个结构存放在由程序自己分配的内存中，所以可以在结构的后面附加一些自定义的数据，这样通过 `_lpSEH` 参数就可以在堆栈中寻址到这些数据。下面是根据这个思路修改后的 SEH 异常处理注册代码：

```

;*****
; 将 _Handler 子程序注册为异常处理程序
;*****
        push ebp                ; 附加数据
        push offset _SafePlace ; 附加数据
        push offset _Handler

```

```

        push fs:[0]
        mov     fs:[0], esp
        ...
        ...
_SafePlace:
        ...
;*****
; 恢复原来的 SEH 链
;*****
        pop     fs:[0]
        add     esp, 0ch

```

在这里，程序在标准的 EXCEPTION_REGISTRATION 结构后面增加了两个自定义数据，一个是“安全地址”，一个是原先的 ebp 值，同时，回调函数也进行了相应的修改：

```

_Handler      proc C _lpExceptionRecord, _lpSEH, \
                _lpContext, _lpDispatcherContext

        pushad
        mov     esi, _lpExceptionRecord
        mov     edi, _lpContext
        assume  esi:ptr EXCEPTION_RECORD, edi:ptr CONTEXT
;*****
; 将 EIP 指向安全的位置并恢复堆栈
;*****
        mov     eax, _lpSEH
        push    [eax + 8]
        pop     [edi].regEip
        push    [eax + 0ch]
        pop     [edi].regEbp
        push    eax
        pop     [edi].regEsp
        assume  esi:nothing, edi:nothing
        popad
        mov     eax, ExceptionContinueExecution
        ret

_Handler      endp

```

将 _lpSEH 参数放入 eax 后，[eax] 相当于 EXCEPTION_REGISTRATION 结构的 prev 字段，[eax+4] 相当于 handler 字段，从 [eax+8] 开始就是程序自定义的数据了，按照主程序中的入栈顺序，[eax+8] 是“安全地址”，[eax+0ch] 是程序保存的 ebp 值，程序中并没有单独设置一个自定义字段来保存 esp，因为在这个例子中，_lpSEH 本身就相当于正确的 esp。经过这样的处理后，整个异常处理过程将不使用任何全局变量。修改后的完整代码可以在本书所附光盘的 Chapter14\SEH02 目录中找到。

Windows 下的许多高级语言都在 EXCEPTION_REGISTRATION 结构的后面添加自定义的数据，比如，Microsoft SDK 的 except.inc 中是这样定义的：

```

__EXCEPTIONREGISTRATIONRECORD struct
    prev_structure      dd      ?
    ExceptionHandler    dd      ?

```

```

        ExceptionFilter      dd      ?      ; 附加数据
        FilterFrame         dd      ?      ; 附加数据
        PExceptionInfoPtrs  dd      ?      ; 附加数据
    _EXCEPTIONREGISTRATIONRECORD ends

```

而在 VC++ 中是这样定义的：

```

struct _EXCEPTION_REGISTRATION{
    struct _EXCEPTION_REGISTRATION *prev;
    void (*handler)(PEXCEPTION_RECORD,
                    PEXCEPTION_REGISTRATION,
                    PCONTEXT,
                    PEXCEPTION_RECORD);
    struct scopetable_entry *scopetable; //附加数据
    int trylevel;                       //附加数据
    int _ebp;                           //附加数据
    PEXCEPTION_POINTERS xpointers;
};

```

除了上面两个以不同方式定义的结构，笔者在很多汇编源代码中也见过更多的各不相同的 EXCEPTION_REGISTRATION 结构定义，正是因为这些结构的定义各不相同，Microsoft 又没有提供一份标准的文档，这使很多初次接触 SEH 的人根本搞不清楚 SEH 究竟是如何定义的。读者现在应该明白这种现象的由来了，回过头去看一看这些结构，就可以发现它们的前面两个字段就是基本的 EXCEPTION_REGISTRATION 结构！

2. 回调函数的返回值

SEH 异常处理回调函数的返回值定义不同于筛选器异常处理回调函数，它可以使用下面列出的 4 种取值。

- ExceptionContinueExecution（等于 0）：回调函数返回后，系统将线程环境设置为 _lpContext 参数指定的 CONTEXT 结构并继续执行。
- ExceptionContinueSearch（等于 1）：回调函数拒绝处理这个异常，系统将通过 EXCEPTION_REGISTRATION 结构的 prev 字段得到前一个回调函数的地址并调用它。
- ExceptionNestedException（等于 2）：回调函数在执行中又发生了新的异常，即发生了嵌套的异常。
- ExceptionCollidedUnwind（等于 3）：发生了嵌套的展开操作（展开操作的介绍参见 14.3.4 节）。

14.3.3 SEH 链和异常的传递

每次定义了一个新的 SEH 异常处理回调函数时，EXCEPTION_REGISTRATION 结构的 prev 字段都被要求填写为原来的 EXCEPTION_REGISTRATION 结构地址，随着应用程序对执行模块的调用一层层深入下去，如果有多个模块设置了回调函数，那么到最后全部的回调函数会形成一个 SEH 链，如图 14.2 所示。

当程序中有多个线程在运行的时候，每个线程中都会存在各自的 SEH 链，这些 SEH 链中指

定了多个回调函数，除它们以外，系统中可能还会存在一个全局性的筛选器异常处理回调函数，再者，如果进程被调试的话，调试器进程也相当于一个异常处理程序存在。既然会同时存在这么多的回调函数，而每个函数都可能对发生的异常提出不同的处理意见，那么当一个异常发生的时候，系统究竟该听谁的意见呢？

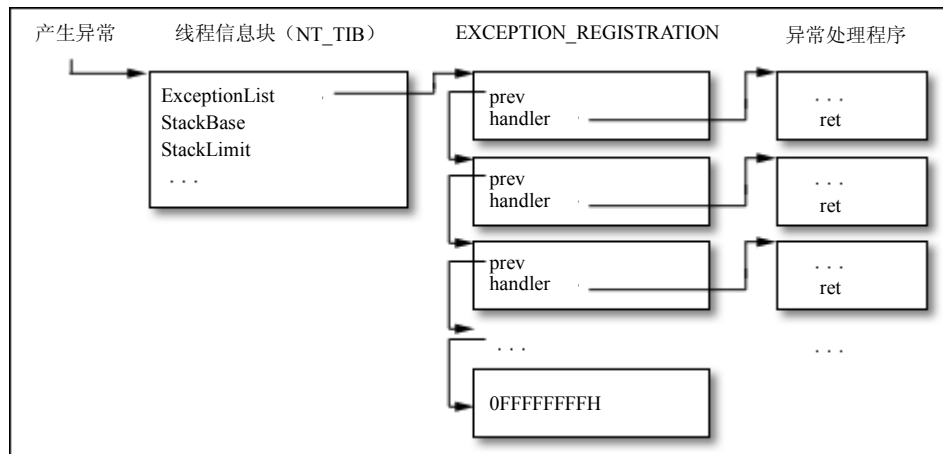


图 14.2 SEH 链

在这种情况下，系统按照一定的步骤选择一个回调函数并执行它，如果这个被执行的回调函数可以处理这个异常，那么程序被修正后继续执行并且其他的回调函数不会再被执行，否则系统继续执行下一个回调函数，查找的步骤如下：

(1) 系统查看产生异常的进程是否正在被调试，如果正在被调试的话，那么向调试器发送 EXCEPTION_DEBUG_EVENT 事件。

(2) 如果进程没有被调试或者调试器不去处理这个异常，那么系统检查异常所处的线程，并在这个线程的环境中查看 fs:[0] 来确定是否安装有 SEH 异常处理回调函数，如果有的话则调用它。

(3) 回调函数尝试处理这个异常，如果可以正确处理的话，则修正错误并将返回值设置为 ExceptionContinueExecution，这时系统将结束整个查找过程。

(4) 如果回调函数返回 ExceptionContinueSearch，告知系统它无法处理这个异常，那么系统将根据 SEH 链中的 prev 字段得到上一个回调函数地址并重复步骤 (3)，直到链中的某个回调函数返回 ExceptionContinueExecution 为止，查找结束。

(5) 如果到了 SEH 链的尾部却没有一个回调函数愿意处理这个异常，那么系统将再次检测进程是否正在被调试，如果被调试的话，则再一次通知调试器。

(6) 如果调试器还是不去处理这个异常或者进程没有被调试，那么系统检查有没有安装筛选器回调函数，如果有，则去调用它，筛选器回调函数返回时，系统默认的异常处理程序根据这个返回值将做相应的动作。

(7) 如果没有安装筛选器回调函数，系统直接调用默认的异常处理程序终止进程。

这个过程归纳起来就是：系统按照调试器、SEH 链上从新到旧的各个回调函数、筛选器回调函数的步骤一个个去调用它们，一直到某个回调函数愿意处理异常为止。如果大家都无法处理异常的话，那么最后由系统默认的异常处理程序来终止发生异常的进程。



Windows 拿着一份处理异常的活挨个问每个回调函数，“你干不干？”，“不干”，“你呢？”，“我也不干”……当问到某一个的时候，他说：“那我来干好了！”，那么 Windows 就不会再问其他人了，于是相安无事。

有时，问完了一圈以后谁都不愿干活，Windows 大怒：“谁都不干，看我炒了你们！”，于是就把整个进程终止掉了，所有的回调函数随之完蛋。

14.3.4 展开操作（Unwinding）

执行上面演示的 SEH 例子文件，程序会在显示了如图 14.3 中 A 所示的消息框后，再显示一个“转移到安全地址”的消息框后正常退出，这一切都在我们的意料之中。

现在来看看回调函数不处理异常时会怎样，将 SEH.asm 修改一下，去掉回调函数中修正 eip 寄存器的指令并将函数的返回值改为 ExceptionContinueSearch，编译执行后再执行一下。首先看到的是图 14.3 中 A 所示的消息框，单击“确定”按钮后，程序不会再显示“转移到安全地址”的消息框，而是出现系统的错误报告对话框，到此为止也在我们的意料之中，现在，单击“确定”按钮，奇怪的事情出现了，回调函数再一次被调用并显示了如图 14.3 中 B 所示的消息框！

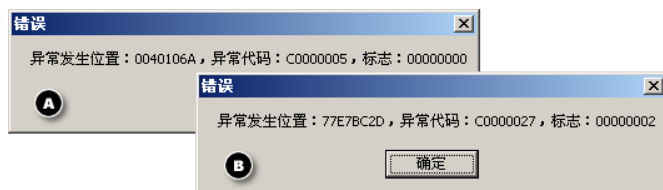


图 14.3 展开操作时的异常代码和标志

进一步试验可以发现，如果程序在 SEH 链上挂了多个回调函数，并且每个回调函数都不处理异常的话，在系统默认的显示错误的对话框出现以后，每个回调函数都会被再调用一遍，这时参数中指定的异常代码是 EXCEPTION_UNWIND，异常标志的取值是 2，也就是 EXCEPTION_UNWINDING 标志。这种调用并不是要求回调函数去处理什么异常，而是告知回调函数：“你将要被卸掉了，自己处理一些后事吧”，在这时回调函数应该进行一些卸载前的扫尾工作并且返回 ExceptionContinueSearch。

对回调函数的这种调用是由展开操作（Unwinding）引起的。当 SEH 链上的某个回调函数进行展开操作时，它所做的事情是从 SEH 链上的第一个回调函数开始（也就是 fs:[0] 指定的回调函数），以 EXCEPTION_UNWIND 代码和 EXCEPTION_UNWINDING 标志去调用每个回调函数，一直到调用到自身所处的位置为止，然后将自身之前的所有回调函数卸载，也就是将 fs:[0] 直接指向描述自身位置的那个 EXCEPTION_REGISTRATION 结构。当进行展开操作后，发起展开操作

的那个回调函数将成为 SEH 链上的第一个回调函数。

1. 为什么要进行展开操作

展开操作在某些情况下是必要的。原因之一是让被卸载的回调函数有机会进行扫尾操作；原因之二是为了防止某些异常情况的发生，这个原因分析起来要复杂一些。

为了程序的模块化设计，一般在堆栈中构造 EXCEPTION_REGISTRATION 结构来注册 SEH 异常处理回调函数，这种方法已经成为各种语言注册 SEH 异常处理程序的首选，然而，与将结构定义成全局变量相比，这种方法又带来了一个新的问题。

现在来看一个典型的应用，如图 14.4 所示，假设在主程序中调用 _Proc1 子程序来实现某种功能，由于这个子程序将涉及内存操作，所以设置了一个回调函数来处理内存访问异常，在 _Proc1 中又会调用 _Proc2 子程序来对所分配的内存中的数据进行一些运算，为了检测计算中的溢出错误，_Proc2 设置了一个回调函数来处理溢出或除零异常，对其他的异常将不予处理并让它在 SEH 链中继续传递。

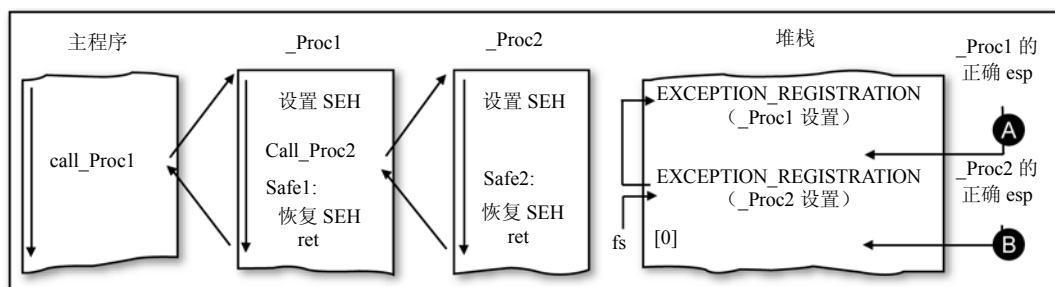


图 14.4 堆栈中的 SEH 链存在情况

当程序执行到 _Proc2 中间的时候，堆栈中的数据如图 14.4 的右边所示，最下方是 _Proc2 注册异常处理回调函数使用的 EXCEPTION_REGISTRATION 结构，现在 fs:[0] 指向这个结构，上面一点是 _Proc2 使用的局部变量、返回地址等，再上面就是注册 _Proc1 中异常处理回调函数使用的 EXCEPTION_REGISTRATION 结构了。

当 _Proc2 中发生溢出异常时，_Proc2 的回调函数将程序修正到 Safe2 执行，在这里堆栈被修正到如图 14.4 中的 B 所示的位置。到 _Proc2 返回的时候，回调函数被卸掉，堆栈中的后一个 EXCEPTION_REGISTRATION 结构被丢弃且 fs:[0] 被恢复指向 _Proc1 设置的结构中，一切都很正常。

但是在 _Proc2 中发生内存存取异常的时候，问题就出现了，这时系统根据 fs:[0] 的值首先找到并调用 _Proc2 设置的回调函数，但这个回调函数不处理这种异常，它会要求 Windows 继续搜索，接下来 _Proc1 设置的回调函数被调用，在这里堆栈被修正到如图 14.4 的 A 所示的位置，这是执行到 Safe1 位置时正确的堆栈位置。

问题就在这里，这时候 esp 指向 A，在 A 位置以下的堆栈空间都是自由的，包括 A 和 B 之间的堆栈空间，如果 _Proc1 接下来进行了一些入栈出栈的操作，原先由 _Proc2 设置的 EXCEPTION_REGISTRATION 结构就会被冲掉，不要忘了这时 fs:[0] 还指向这个失效的结构，如

果这时再发生一个异常的话，Windows 就会调用一个无效的回调函数地址。

这就是要进行展开操作的第二个原因，为了防止这个意外，_Proc2 的异常处理回调函数在被执行的时候应该将 fs:[0] 中的值重新置为本身使用的 EXCEPTION_REGISTRATION 结构的地址，这样即使再次发生异常，也不会有前面这种危险的情况发生，这个操作相当于将后面所设的所有异常处理回调函数都卸掉了。

2. 完整的异常处理回调函数

综上所述，一个完整的异常处理回调函数应该包括异常处理、展开操作和响应展开调用等部分，其结构示意图如下所示：

_Handler	proc C	_lpExceptionRecord, _lpSEH, \
		_lpContext, _lpDispatcherContext
	.if	(异常代码 == 0c0000027h) \
		(异常标志 & EXCEPTION_UNWINDING) \
		(异常标志 & EXCEPTION_UNWINDING_FOR_EXIT)
		进行释放资源等扫尾工作 ; (1)
		mov eax, ExceptionContinueSearch
	.elseif	异常代码 == 可以处理的异常代码
		处理异常，对 CONTEXT 进行修正 ; (2)
		进行展开操作 ; (3)
		mov eax, ExceptionContinueExecution
	.else	; 其他无法处理的异常代码
		mov eax, ExceptionContinueSearch ; (4)
	.endif	
	ret	
_Handler	endp	

但是在实际的应用中，并不一定要存在上面所示的全部代码，如果某个异常处理回调函数对所有的异常代码都进行处理的话，那么就不会有（4）所示的代码，这样在它以后的回调函数就不可能再被调用，这样一来，这个回调函数也不可能被其他回调函数以展开操作的异常代码调用，结果是（1）所示的代码也就不需要了。

另外，当回调函数能够确定自己是 SEH 链上的最后一个回调函数的话，由于不存在展开操作的对象，也就不需要（3）所示的代码。

在本章的 SEH 例子中，程序设置的回调函数既是最后一个异常处理回调函数，又对所有的异常代码进行处理并将程序转移到“安全地址”去执行，所以仅仅需要（2）所示的代码。

3. 如何进行展开操作

自己书写展开操作的代码并不复杂，第一步是在一个循环中以 EXCEPTION_UNWINDING 标志调用从 fs:[0] 开始到当前回调函数为止的所有回调函数，第二步是将 fs:[0] 重新设置一下，指向注册当前回调函数使用的 EXCEPTION_REGISTRATION 结构就可以了。

但是，更方便的办法是使用 Win32 中未公开的函数 RtlUnwind，这个函数可以完成上述的功能，函数的使用方法如下所示：

invoke RtlUnwind, lpLastStackFrame, lpCodeLabel, lpExceptionRecord, dwRet

使用参数 lpLastStackFrame 可以有两种方法。

第一，将它指定为当前回调函数使用的 EXCEPTION_REGISTRATION 结构地址的话，表示对当前回调函数之后的所有其他回调函数进行展开操作，RtlUnwind 函数调用每个被展开的回调函数时，异常标志中会含有 EXCEPTION_UNWINDING 标志位。

第二，如果这个参数指定为 NULL 的话，表示对 SEH 链上所有的回调函数进行展开操作，这时所有回调函数参数中的异常标志在带有 EXCEPTION_UNWINDING 标志位的同时也带有 EXCEPTION_UNWINDING_FOR_EXIT 标志位，这种方式的展开称为退出展开 (Exit Unwind)。

lpCodeLabel 指定函数返回的位置。如果这个参数指定为 NULL，函数使用正常的返回方式，也就是返回到调用 RtlUnwind 函数的后面一条指令，否则，函数直接返回到 lpCodeLabel 指定的地址。

lpExceptionRecord 指定一个 EXCEPTION_RECORD 结构。这个结构将在展开操作的时候被传给每一个被调用的回调函数，一般建议使用 NULL 来让系统自动生成代表展开操作的 EXCEPTION_RECORD 结构。dwRet 参数一般不被使用，可以将它指定为 NULL。

本书所附光盘的 Chapter14\Unwind 目录中包含了一个 SEH 展开操作的例子，读者可以自行分析一下，由于篇幅所限，在此就不再列出了。



使用 RtlUnwind 函数时要注意的是：这个函数并不像其他 API 函数一样保存 esi, edi 和 ebx 寄存器，在函数返回的时候这些寄存器的值可能会被改变，所以，如果程序用到了这些寄存器的话，必须自己去保存和恢复它们。



最后需要说明的是，SEH 异常处理属于 Win32 中未公开的特征，本章中的大部分内容无法从 Microsoft 的正式文档中查到，它们来自于各种零星的资料（包括笔者对一些例子的分析以及编程测试的结果），所以可能与其他资料有所出入。如果读者发现存在错误或者有什么疑问，请告知笔者。

* 非常感谢温玉杰 (Hume) 提供本章中 SEH 相关内容的帮助！

第 15 章

注册表和 INI 文件

15.1 注册表和 INI 文件简介

在一个操作系统中，无论是操作系统本身还是运行于其中的大部分应用程序，都需要使用某种方式保存配置信息。在 DOS 系统中，配置信息往往是软件的开发者根据自己的喜好用各种途径加以保存的，比如在磁盘上面写一个二进制的 .dat 文件，或者写一个文本文件等，这些配置文件中数据的格式也是各不相同的。

到了 Windows 3.x 系统中，这种情况发生了变化，虽然软件的开发者还可以沿用这种自成体系的方法，但是系统也提供了一种标准的初始化文件格式（Initialization File）来保存配置信息。初始化文件是一种以 INI 为扩展名的文本文件，操作系统提供了一套专用的函数对 INI 文件进行操作。Windows 3.x 不仅鼓励程序员使用 INI 文件，操作系统本身也使用 INI 文件来保存配置信息，比如，Windows 安装目录下的 Win.ini 文件中保存了桌面设置和与应用程序运行有关的信息；System.ini 文件中保存了与硬件配置有关的信息，另外，Control.ini 与 Program.ini 等文件也是很重要的配置文件。

INI 文件在使用中存在诸多缺陷。由于 INI 文件是文本文件，使用任何文本编辑器都可以随意对它进行修改，所以安全性不是很好。另外，INI 文件的结构比较简单，无法保存格式复杂的数据，如很长的二进制数据或带回车的字符串等。最主要的缺点还是单个 INI 文件的大小不能超过 64 KB。如果不同的应用程序都将自己的配置信息保存于 Win.ini 或者 System.ini 中，那么这些文件的规模很快就会超过限制的长度。如果不同应用程序都使用自己的 INI 的话，那么集中管理又成了一个问題。

在 Windows 9x 和 Windows NT 系列操作系统中，改用了一种全新的方法来管理配置信息，那就是使用集中管理的注册表（Registry）。注册表并不像它的中文名称所称的那样是“表”，而是一种格式由系统定义的数据库，它存放于某些二进制文件中。不同操作系统中对应的文件名可能有所不同，比如，Windows 9x 系统中的注册表文件由位于 Windows 安装目录中的 System.dat 和 User.dat 两个文件组成，而 NT 系统的注册表往往由位于 Windows 安装目录下的 System32\Config 目录中的多个文件构成。操作系统将这些不同的文件集中“虚拟”成整个注

册表供系统自身及应用程序使用。

与 INI 文件对于 Windows 3.x 系统的重要性相比，注册表对于 Windows 9x 和 NT 系统来说显得更加重要。因为绝大多数系统使用的配置信息都存放于此，如系统的硬件配置、安装的驱动程序列表、文件的关联信息、系统的网络配置和权限配置等，这些配置信息直接关系到 Windows 系统的启动和初始化过程。如果注册表受到了破坏，那么轻者 Windows 的启动过程出现异常，重者可能会使整个系统无法启动。另外，大部分应用程序的配置信息也存放于此，如果注册表受到了破坏，即使操作系统能正常启动，应用程序的运行也可能会受影响。

正是因为注册表结构的封闭性和重要性，我们无法再使用像编辑 INI 文件那样的简单方法来编辑注册表文件。实际上，Windows 系统对注册表文件的保护很严格，当系统在运行的时候，注册表文件是被操作系统以独占方式打开的，其他应用程序即使是用最基本的读权限也无法打开它们，更不用说对它们进行写操作了。

要对注册表进行操作，必须使用系统提供的接口。Windows 为此提供了一系列的注册表操作函数，应用程序可以通过它们来完成注册表编辑器（Regedit 程序）能完成的全部功能，甚至包括远程操作注册表以及对 .reg 文件进行导入和导出等操作。

为了提供向下兼容性，Windows 9x 和 NT 系统在支持注册表操作的同时也支持 INI 文件的操作。实际上对于某些“Copy and play”的小程序来说，需要保存的配置信息并不复杂，使用 INI 文件可能更加简单实用，而且保存于注册表中的配置信息是无法随文件拷贝到其他计算机中的。如果某些应用程序希望拷贝程序的同时可以拷贝配置信息，那么最好还是使用 INI 文件，所以在 Windows 9x 和 NT 系统中，INI 文件的使用还是相当广泛的。

本章用两个单独的例子，详细介绍 INI 文件和注册表的使用方法。

15.2 INI 文件的操作

15.2.1 INI 文件的结构

INI 文件是一种文本格式的文件，其中的数据组织格式为：

```
;注释
[Section1 Name]
KeyName1=value1
;注释
KeyName2=value2
...
[Section2 Name]
KeyName1=value1
KeyName2=value2
...
```

INI 文件中可以存在多个小节（Section），每个小节的开始用包括在一对方括号中的小节名称指定，不同的小节不能重名，一个小节的内容从小节名称的下一行开始，直到下一个小节开始为止。用户程序可以按照自己的需求建立多个小节。

在每个小节中可以定义多个键（Key），每一个键由一个“键名=键值”格式的字符串组成，并独自占用一行。在同一个小节中不能存在同名的键，但是在不同的小节中可以存在同名的键。

如果需要在 INI 文件的某些地方加注释，可以将注释放在单独的一行中，行首以分号开始，注释行出现的地方并没有什么限制，既可以出现在文件的最前面，也可以出现在文件的任何一行中，如上面的例子中注释出现在小节名称的前面以及两个键的中间。

一般来说，如果在自己开发的应用程序中使用系统定义的 INI 文件，如 Win.ini 等，由于文件中已经存在多个小节，那么自己建立一个独立的小节比较合适，然后在这个小节中定义不同的键值，比如，下面是笔者的计算机上 Win.ini 文件的片断：

```
...
[MCI Extensions]
asf=MPEGVideo
asx=MPEGVideo
m3u=MPEGVideo
mp2v=MPEGVideo
mp3=MPEGVideo
mpv2=MPEGVideo
wma=MPEGVideo
wmv=MPEGVideo

[Hex Workshop]
Path=C:\PROGRA~1\BREAKP~1\HEXWOR~1.1\hworks32.exe
CurrentVersion=3.11
...
```

其中的“MCI Extensions”小节是 Windows 系统自身使用的小节，Windows 在这里定义了一些媒体文件的关联方式，而“Hex Workshop”小节是安装了十六进制编辑器 HexWorkshop 后由这个软件创建并使用的，该软件在小节中用“Path”键定义了软件的安装目录、在“CurrentVersion”键中定义了软件的版本号。

如果觉得往系统 INI 文件中写数据显得不是那么“绿色环保”，那么应用程序可以建立一个独立的 INI 文件。如本节的例子文件就在自己运行的目录中建立了一个 Option.ini 文件，并在“Windows Position”小节的“X”，“Y”键中保存窗口的位置，以便在下次运行的时候将窗口移动到上一次退出时所处的位置，内容如下：

```
[Windows Position]
X=194
Y=162
...
```

Windows 系统提供了一系列函数对 INI 文件进行操作，其中包括读取和设置键值，获取小节名称列表及获取和删除整个小节内容等函数。下面的例子演示了这些功能的使用方法。

15.2.2 管理键值

本节的例子程序存放在所附光盘的 Chapter15\ini 目录中，运行后的界面如图 15.1 所示。例子程序在运行目录下创建了一个 Option.ini 文件，程序中的所有操作都是针对这个文件进行的。当用户在“Section”一栏中输入小节名称、在“Key”一栏中输入键名后，如果 INI 文件中对应的小节和键定义是已经存在的，那么按下“读取 Key”按钮后就会将键值读取到“Value”一栏中；而按下“删除 Key”按钮的时候，可以将这个键删除。

在输入小节和键名后继续在“Value”一栏中输入一个字符串，并按下“保存 Key”按钮，如果指定键已经存在，那么程序用新的键值替换原来的键值；如果键名不存在，则程序创建这个键；如果创建键的时候小节名是不存在的，那么程序在创建键值之前会自动创建小节；在最极端的情况下，当 INI 文件也不存在的时候，那么程序也会创建 INI 文件。



图 15.1 INI 文件操作例子的运行界面

当用户按下“删除 Key”按钮将一个小节中的键逐一删除直到全部键都被删除的时候，小节名称并不会被删除，INI 文件中还会留有一个空的小节名称，按下“删除 Section”按钮后可以将小节名称删除。如果在小节中尚包含有键的时候按下“删除 Section”按钮，那么小节名称包括小节中的全部键都会被删除。

每次进行操作后，程序自动将 INI 文件中的所有小节和键值枚举一遍并将内容显示在图 15.1 下面的编辑框中，以便观察操作的结果。下面通过分析这个程序来了解这些功能的实现方法。

源文件目录中的 Ini.rc 文件定义了如图 15.1 所示的对话框，代码如下：

[illegible]

汇编源文件 `Ini.asm` 的内容如下：

哭

528

枚举小节和键值的功能是在 EnumINI 子程序中完成的。程序在一开始执行的时候或者在每

次收到 WM_COMMAND 消息的时候都调用这个函数，所以每次用户有所操作，INI 文件的变化就会马上在窗口中的编辑框中反映出来。

1. 键值的创建和删除

当按下“保存 Key”按钮时，例子程序使用 WritePrivateProfileString 函数保存键值，这个函数可以往指定的 INI 文件中写入键值，函数的用法是：

```
invoke    WritePrivateProfileString, lpAppName, lpKeyName, lpString, lpFileName
```

在函数的参数中，lpAppName 参数指向包含 Section 名称的字符串，lpKeyName 参数指向包含键名称的字符串，lpString 参数指向键值字符串，最后一个参数指向 INI 文件名字符串。这些字符串都是以 0 字符结束的。

当这些参数全部指定为字符串的时候，函数将在指定 INI 文件的指定小节中写入“键名=键值”格式的行；当指定的 INI 文件、文件中的小节和小节中的键名都已经存在的时候，函数用新键值替换原来的键值；当指定的 INI 文件存在而小节不存在的时候，函数自动创建小节并将键写入；如果连指定的 INI 文件也不存在的话，函数会自动创建文件。总之，程序不必考虑 INI 文件是否存在，小节是否存在或键值定义是否存在等情况，只要调用 WritePrivateProfileString 函数就可以保证配置信息被正确保存。

WritePrivateProfileString 函数也可以用来删除键或者小节，当 lpAppName 和 lpKeyName 参数指定了小节名称和键名，而 lpString 参数指定为 NULL 的时候，函数将指定的键删除，如例子文件中对“删除 Key”按钮的操作就是这样的：

```
.elseif ax == IDC_DEL_KEY
    invoke WritePrivateProfileString, addr @szSection, \
        addr @szKey, NULL, addr szProfileName
```

但是使用这种方法逐一将某个小节中的键全部删除时，空白小节的定义字符串“[SectionName]”还保留在 INI 文件中。如果想要将小节的定义字符串连同小节的全部键定义全部删除的话，可以将 lpKeyName 和 lpString 参数全部指定为 NULL，而 lpAppName 参数指定要删除的小节，如例子文件中对“删除 Section”按钮的处理代码：

```
.elseif ax == IDC_DEL_SEC
    invoke WritePrivateProfileString, addr @szSection, \
        NULL, NULL, addr szProfileName
```

如果函数执行成功，将返回一个非 0 的值，如果执行失败将返回 0。在定义键名的时候，注意不用在名称字符串中包括“=”号，因为等号被用来分隔键名和键值，键名也不能以注释字符“;”开始。在定义键值的时候可以使用等号和分号，但注意不要将键值定义为多行的文本。如果在字符串中包含换行和回车，比如将键值字符串指向下列所示的一个字符串：

```
"hello,world!",0dh,0ah,"this is the second line",0
```

那么函数会成功地被调用，但是最后的 INI 文件中会出现这样的内容：

```
[Section]
Key=hello,world!
```

```
this is the second line
```

显然，函数不加判断地将换行和回车也写到了 INI 文件中，但是当取回键值的时候，只有第一行能被正确取回，而底下的行将当做格式错误的“垃圾”留在 INI 文件中。

由于 INI 文件是以文本方式保存的，所以实际上键值也只能用字符串方式表示，如果需要保存一个数值类型的值，那么程序需要自己使用 `wsprintf` 函数将数值转换成字符串后再保存。比如例子程序在退出时为保存窗口位置，就是在 `SavePosition` 子程序中首先用 `GetWindowRect` 函数获取窗口位置，然后使用 `wsprintf` 函数转换后再保存的。

2. 获取键值

获取键值的操作比较方便，因为这时既可以用 `GetPrivateProfileString` 函数获取键值字符串，也可以使用 `GetPrivateProfileInt` 函数让 Windows 将键值字符串转换成数值后再返回，就像使用 `GetDlgItemInt` 函数转换对话框子窗口中的字符串一样。（比较奇怪的是保存键值的时候并没有一个 `WritePrivateProfileInt` 函数，结果每次还要首先使用 `wsprintf` 函数！）

`GetPrivateProfileString` 函数的用法是：

```
invoke  GetPrivateProfileString, lpAppName, lpKeyName, lpDefault, \
        lpReturnedString, nSize, lpFileName
```

该函数的几个参数与 `WritePrivateProfileString` 的参数类似，也是使用 `lpAppName`，`lpKeyName` 和 `lpFileName` 参数分别指定小节名称、键名和 INI 文件名，但是其余几个参数则有所不同：

`lpReturnedString` 参数指向一个缓冲区，函数在这里返回获取的键值字符串，缓冲区的长度用 `nSize` 参数指定，当缓冲区的长度太小以至于无法容纳返回的字符串时，字符串会被截止到 `nSize-1` 的长度后返回，余下的一个字节用来存放一个 0 字符用做结尾。

`lpDefault` 参数指向一个默认字符串，当指定的键无法找到的时候，函数将这个字符串拷贝到返回缓冲区中。

`GetPrivateProfileString` 还有两种特殊用法：首先，当 `lpAppName` 参数指定为 `NULL` 的时候，函数在缓冲区中返回的是全部小节名称的列表，每个小节名以 0 结尾，全部的名称列表再以一个附加的 0 结束，返回到缓冲区中的数据格式如下所示：

```
小节名称 1, 0, 小节名称 2, 0, ..., 小节名称 n, 0, 0
```

另外，当 `lpAppName` 参数指定了小节名称，而 `lpKeyName` 参数指定为 `NULL` 的时候，函数在缓冲区中返回该小节的全部键名列表，每个键名以 0 结尾，全部列表后面再以一个附加的 0 结束，如下所示：

```
键名 1, 0, 键名 2, 0, ..., 键名 n, 0, 0
```

所以用这两种方法调用 `GetPrivateProfileString` 函数可以实现枚举小节名称和枚举键名的功能。

不管用何种方式使用 `GetPrivateProfileString` 函数，函数的返回值是返回到缓冲区中的

字符串长度（长度中并不包括结尾的 0 字符）。

如果保存的键值是全部由数字字符组成的话（比如例子程序的 _SavePosition 子程序中用 `wsprintf` 函数从窗口位置的坐标值转换过来的字符串），那么可以使用 `GetPrivateProfileInt` 函数直接将字符串转换成数值后再返回：

```
invoke    GetPrivateProfileInt, lpAppName, lpKeyName, nDefault, lpFileName
```

其中 `lpAppName`, `lpKeyName` 和 `lpFileName` 参数的使用方法同上，函数将指定的键值字符串转换成数值类型以后返回，但是函数不支持负数，如果键值字符串是“- 1234”格式的负数，那么函数的返回值是 0。`nDefault` 指定一个默认数值，如果指定的键名不存在的话，函数返回 `nDefault` 指定的数值。

例子程序在初始化的时候，在 `_GetPosition` 子程序中使用 `GetPrivateProfileInt` 函数读出上次退出时保存的窗口位置，并使用 `SetWindowPos` 函数将窗口移动到这个位置上。

15.2.3 管理小节

在键名已知的情况下固然可以使用 `GetPrivateProfileString` 等函数获取键值，但在某些情况下并不是所有的键名都是已知的，比如一个编辑文件需要保存近来编辑过的文件名列表，它可以建立一个小节如下：

```
[History]
file1=C:\My documents\Readme.txt
file2=D:\MyApp\Help.txt
file3=C:\download\win32asm.txt
...
```

这时小节中键的数量和名称就是不定的。另外，也有小节的数量和名称不定的情况，这时就需要对小节或键进行枚举。上一节中已经介绍了在 `GetPrivateProfileString` 函数中通过将 `lpAppName` 或 `lpKeyName` 参数设置为 `NULL` 来获取小节名称列表和键名列表的方法。实际上，Windows 中还有专门用来实现此功能的函数，这些函数不仅可用来枚举小节和键，也可以用来一次性修改整个小节的内容。

`GetPrivateProfileSectionNames` 函数可以用来返回全部小节名称的列表：

```
invoke    GetPrivateProfileSectionNames, lpBuffer, nSize, lpFileName
```

`lpFileName` 参数指向 INI 文件名，`lpBuffer` 参数是一个指针，指向用来返回小节名称列表的缓冲区，`nSize` 参数指定缓冲区的长度。返回到缓冲区中的数据格式也是：“小节名称 1, 0, 小节名称 2, 0, ..., 小节名称 n, 0, 0”的格式。当缓冲区太小以至于不能容纳全部数据的时候，后面的数据被丢弃，全部数据被截尾到 `nSize-2` 的长度，剩下的两个字节用来保存两个表示结束的 0 字符。函数的返回值是返回到缓冲区中的数据长度（不包括结尾的 0 字符）。

`GetPrivateProfileSection` 函数则可以用来返回整个小节的键定义，与上一节介绍的调用 `GetPrivateProfileString` 函数时将 `lpKeyName` 参数设置为 `NULL` 以获取键名列表不同，`GetPrivateProfileSection` 函数返回的是键定义列表。函数的用法是：

invoke	GetPrivateProfileSection, lpAppName, lpBuffer, nSize, lpFileName
--------	--

函数的 lpAppName 指向一个包含小节名称的字符串，返回到缓冲区中的数据格式为：

键名 1=键值 1, 0, 键名 2=键值 2, 0, ..., 键名 n=键值 n, 0, 0

所以，使用这个函数可以同时完成枚举键名和键值的功能。例子程序中就是使用它来枚举键值的，但使用中如果觉得自己处理“键名=键值”字符串来分解键名和键值比较麻烦的话，可以用 GetPrivateProfileString 函数枚举键名并再次调用它获取指定键的键值。

在 Windows 9x 中，缓冲区最大不能超过 32 767 B，而在 Windows NT 中则没有限制，函数的返回值是返回到缓存区中的数据长度（不包括结尾的 0 字符）。

WritePrivateProfileSection 函数则将“键名 1=键值 1, 0, 键名 2=键值 2, 0, ..., 键名 n=键值 n, 0, 0”格式的小节数据一次性全部写入。函数的用法是：

invoke	WritePrivateProfileSection, lpAppName, lpString, lpFileName
--------	---

lpString 参数指向包含键值定义列表的缓冲区，函数执行后，指定小节原来的键定义被全部删除，然后加入新的键定义。如果执行成功，函数返回非 0 值，否则返回 0。

15.2.4 使用不同的 INI 文件

在前面介绍的这些 INI 文件函数中，当 lpFileName 参数指定的文件名字符串中不包括路径时，系统将认为文件位于 Windows 安装目录下，这样当函数创建 INI 文件的时候，就会把文件创建于 Windows 安装目录下。但是大部分情况下，希望 INI 文件位于程序的运行目录下，这样拷贝文件的时候可以连同 INI 文件一起拷贝，另外，在卸载或删除程序的时候可以避免在 Windows 目录中留下一个“垃圾”INI 文件。

如果希望在程序的运行目录而非 Windows 目录中建立 INI 文件，那么最简单的方法就是在 INI 文件名前面加上“.\”路径，也就是说把文件名写成“.\MyIniFile.ini”的格式，这样系统会在当前目录下建立 INI 文件。这种方法的缺陷是如果程序运行中需要不断切换当前目录的话，系统就会在不同的地方乱建 INI 文件。一个常见的情况就是使用“打开文件”通用对话框的时候，系统会将当前目录切换到对话框中浏览的那个目录，这样写 INI 文件时就会造成 INI 文件的位置根本无法确定。一个相对保险的办法就是像例子程序中所示的那样，在程序一开始运行的时候就使用 GetCurrentDirectory 函数获取当前目录，然后将 INI 文件名添加到目录后组成一个全路径的文件名，以后在所有的操作中都使用这个文件名。而最保险的办法就是首先用 GetModuleFileName 获取包含全路径的当前执行文件名，再从这个文件名中分离出路径，最后添加上 INI 文件名。

如果要操作的是 Windows 安装目录下的 Win.ini 文件而非其他 INI 文件时，那么既可以使用上面这些函数，也可以使用另一组专门用于操作 Win.ini 文件的函数，这组函数是：

invoke	GetProfileString, lpAppName, lpKeyName, lpDefault, lpBuffer, nSize
invoke	GetProfileInt, lpAppName, lpKeyName, nDefault
invoke	WriteProfileString, lpAppName, lpKeyName, lpString
invoke	GetProfileSection, lpAppName, lpBuffer, nSize

invoke WriteProfileSection, lpAppName, lpString

可以看出，与操作通用 INI 文件的函数相比，这组函数的函数名中少了“Private”单词，参数中少了 lpFileName 参数（因为操作的 INI 文件名就是 Win.ini，并不需要单独指定），所有其他参数的用法都是相同的。

读者可以注意到这些函数中似乎少了一个 GetProfileSectionNames 函数，实际上并没有这个函数，如果要枚举 Win.ini 文件中的小节列表的话，只需把 GetPrivateProfileSectionNames 函数的 lpFileName 参数指定为 NULL 即可，并不需要单独设置一个类似于 GetProfileSectionNames 的函数。

15.3 对注册表的操作

注册表在 Windows 9x 及 NT 系统中是很重要的，不谈操作系统本身对注册表依赖性的大小，仅从应用程序的角度来说，除了用来代替 INI 文件用做保存配置信息以外，有些功能是必须通过操作注册表来完成的，比如，要让一个程序在 Windows 启动的时候自动运行，那就必须在注册表的 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run 子键下面添加一个键值定义；如果要让应用程序和某种数据文件相关联，就必须在注册表的 HKEY_CLASSES_ROOT 键下面为相应的数据文件扩展名设置关联信息；再比如，编写好一个 COM 组件以后，要为组件的 DLL 文件在注册表中添加注册信息，组件才能被其他程序使用。另外，操作系统本身将绝大多数的配置信息保存于注册表中，应用程序在运行中往往需要查询这些系统信息来决定运行的方式。

要自如地使用注册表，必须解决两方面的问题，第一是如何在程序中对注册表进行读写与枚举等操作；第二就是要搜集整理关于注册表键值定义的资料。因为注册表是 Windows 中出了名的“数据迷宫”，很多键的定义只有 Microsoft 自己知道，公开的键定义也足够编成一本手册了，仅知道如何读写注册表却不知道该怎么往哪里读写是没用的。由于本节并不是当做注册表的定义手册来写的，所以内容仅涉及第一个方面的问题。本节讨论的内容是：注册表的数据组织方式，以及如何用 Win32 汇编来读写和枚举注册表的内容。

15.3.1 注册表的结构

1. 注册表的数据组织方式

INI 文件中的数据是按照两层组织的——只能通过一些在结构上“平行”的小节来归类不同的键，这就像一个驱动器中只能建立一层目录来管理文件一样非常不便，与此相比，注册表的结构有很大改进。

注册表中的数据是分多个层次来组织的，组织的方式类似于磁盘目录的多层组织方式。与文件系统中根目录、子目录和文件这样的层次划分类似，注册表中的数据层次分为根键、键和键值项，其中根键就相当于文件系统中的根目录，键相当于子目录，键值项相当于文件。根键和子键是为了将不同的键值项分类组织而定义的，只有键值项中才包含真正的配置数据。

在 Windows 自带的注册表编辑器 Regedit 中就可以看出注册表的结构来。如图 15.2 所示，

注册表中的根键有 6 个，其名称是 Windows 规定的，并且是固定不变的，它们分别是 HKEY_CLASSES_ROOT，HKEY_CURRENT_USER，HKEY_LOCAL_MACHINE，HKEY_CURRENT_CONFIG，HKEY_DYN_DATA 和 HKEY_USERS。在每个根键下可以建立不同的键，以 HKEY_LOCAL_MACHINE 根键为例，下面有 HARDWARE，SAM，SECURITY 和 SOFTWARE 等子键，而 SOFTWARE 键下面又由各种应用程序创建了一些子键，如 ACD Systems，Acer 和 Adobe 等，键和子键的层次关系是相对的，就像一个目录既可以是其上层目录的子目录，又可以是其下层目录的父目录一样。

在一个键中既可以继续建立多个子键，也可以同时建立多个键值项，就像一个目录中既可以建立多个子目录，同时也可以存放多个文件一样。图 15.2 右边窗口中列出的就是 ACDSee 键中定义的键值项。

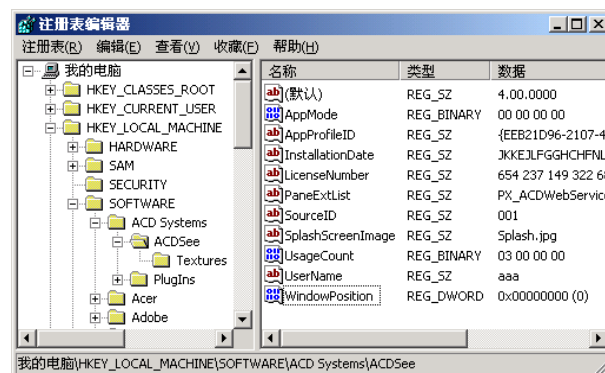


图 15.2 注册表的结构

每个键值项由键值名称和键值数据组成（就像是文件名和文件中的数据的关系），如上图键值名称 AppMode 中的数据是“00 00 00 00”、键值名称 UserName 中的数据是“aaa”。不像 INI 文件中的键值只能定义为字符串，注册表键值的数据类型要丰富得多，全部可用的键值类型如表 15.1 所示。

表 15.1 注册表的键值类型

键值类型	说 明
REG_BINARY	任何方式的二进制数据
REG_DWORD	一个 32 位的双字（同 REG_DWORD_LITTLE_ENDIAN）
REG_DWORD_BIG_ENDIAN	高位排在低字节的双字
REG_EXPAND_SZ	扩展字符串，可以将中间类似于“%PATH%”类型的子串按照环境变量中的定义值扩展
REG_LINK	Unicode 符号链接
REG_MULTI_SZ	多字符串，格式为“字符串 1, 0, 字符串 2, 0, 0”类型
REG_RESOURCE_LIST	设备驱动程序资源列表
REG_SZ	以 0 结尾的字符串（最常用的类型！）

图 15.2 中的 AppMode 和 UsageCount 键值项是 REG_BINARY 类型的，WindowPosition 键值项是 REG_DWORD 类型的，其余的键值项是 REG_SZ 类型的，这 3 种类型的键值项在注册表中最常见的。每个键下面还可以有一个没有名称的键值项，称为默认键，默认键必须是 REG_SZ

或 REG_EXPAND_SZ 类型的。

2. 注册表中的根键

注册表的结构中大量采用“映射”关系，系统定义的 6 种根键其实存放在不同的文件中。在 Windows 9x 系统中，HKEY_LOCAL_MACHINE 根键的内容存放在 System.dat 文件中，HKEY_USERS 根键的内容存放在 User.dat 文件中。而在 Windows NT 系统中，注册表的内容存放得更分散，连 HKEY_LOCAL_MACHINE 根键中的不同子键 SOFTWARE，SAM，SECURITY 和 SYSTEM 等都分开存放在 Windows\system32\config 目录下的不同文件中。

HKEY_LOCAL_MACHINE 和 HKEY_USERS 根键是注册表中的两大根键，其余的根键都是它们的派生，实际上它们都是这两大根键下面某些子键的映射，如 HKEY_CLASSES_ROOT 根键是 HKEY_LOCAL_MACHINE 根键下 SOFTWARE\Classes 子键的映射，HKEY_CURRENT_CONFIG 根键是 HKEY_LOCAL_MACHINE 根键下 Config 子键的映射。

HKEY_USERS 根键中的内容是用户配置信息，其内容取决于计算机是否激活了用户配置文件。若未激活用户配置文件，则里面只有名为.DEFAULT 的单一子键，该子键包括与所有用户相关的各种设置。若激活了用户配置文件并且正确地执行了登录操作，则根键下还会有代表当前登录用户的子键，这时候 HKEY_CURRENT_USER 根键就是这个子键的映射。

由于 HKEY_DYN_DATA 保存了系统运行时的动态数据，它反映出系统的当前状态，所以它的数据在每次运行时都是不一样的。应用程序一般不使用这个根键。

对这些实际上是其他数据的映射的根键来说，操作根键上的数据和操作未经映射前的数据产生的效果是一样的，系统建立映射可以让键值数据的组织更清晰，操作起来更加快捷、方便。

15.3.2 管理子键

本节用一个例子来演示对注册表的操作方法，例子代码在所附光盘的 Chapter15\Reg 目录中，运行后的界面如图 15.3 所示。例子程序演示了子键的创建和删除，键值项的创建、读取和删除，以及枚举子键和键值项的功能。

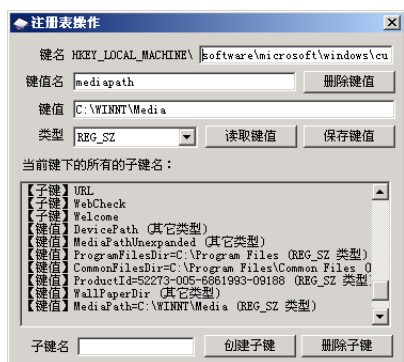


图 15.3 注册表操作例子的运行界面

例子程序对 HKEY_LOCAL_MACHINE 根键进行操作，当在“键名”一栏中输入子键名称字符串，并在“键值名”一栏输入键值名称后，按下“读取键值”按钮，如果指定子键中存在这个键值项，程序会读出键值数据并显示在“键值”一栏中；如果按下“删除键值”按钮，那么对应键值项被删除；读者也可以在“键值”一栏中输入其他数据并选定“类型”，然后按下“保存键值”按钮将新的键值数据设置到注册表中。

例子程序也可以对子键进行操作。在对话框最下面的“子键名”一栏中输入子键名称，按下“创建子键”的话，函数会在“键名”一栏指定的键下面创建子键，如果按下“删除子键”按钮，那么“键名”指定的键下面由“子键名”指定的子键会被删除。

器印

器印

器印

器印

538

```

        mov     @dwIndex, 0
        invoke   SetDlgItemText, hWinMain, IDC_KEYLIST, NULL
;*****
; 枚举子键
;*****
invoke   RegOpenKeyEx, HKEY_LOCAL_MACHINE, _lpKey, NULL, \
        KEY_ENUMERATE_SUB_KEYS, addr @hKey
.if     eax == ERROR_SUCCESS
        .while  TRUE
            mov     @dwSize, sizeof @szBuffer
            invoke   RegEnumKeyEx, @hKey, @dwIndex, addr @szBuffer, \
                addr @dwSize, NULL, NULL, NULL, NULL
            .break   .if eax == ERROR_NO_MORE_ITEMS
            invoke   wsprintf, addr @szBuffer1, \
                addr szFmtSubkey, addr @szBuffer
            invoke   SendDlgItemMessage, hWinMain, IDC_KEYLIST, \
                EM_REPLACESEL, 0, addr @szBuffer1
            inc      @dwIndex
        .endw
        invoke   RegCloseKey, @hKey
    .endif
;*****
; 枚举键值
;*****
mov     @dwIndex, 0
invoke   RegOpenKeyEx, HKEY_LOCAL_MACHINE, _lpKey, NULL, \
        KEY_QUERY_VALUE, addr @hKey
.if     eax == ERROR_SUCCESS
        .while  TRUE
            mov     @dwSize, sizeof @szBuffer
            mov     @dwSize1, sizeof @szValue
            invoke   RegEnumValue, @hKey, @dwIndex, addr @szBuffer, \
                addr @dwSize, NULL, addr @dwType, \
                addr @szValue, addr @dwSize1
            .break   .if eax == ERROR_NO_MORE_ITEMS
            mov     eax, @dwType
            .if     eax == REG_SZ
                invoke   wsprintf, addr @szBuffer1, \
                    addr szFmtSz, addr @szBuffer, addr @szValue
            .elseif eax == REG_DWORD
                invoke   wsprintf, addr @szBuffer1, addr szFmtDw, \
                    addr @szBuffer, dword ptr @szValue
            .else
                invoke   wsprintf, addr @szBuffer1, \
                    addr szFmtValue, addr @szBuffer
            .endif
            invoke   SendDlgItemMessage, hWinMain, IDC_KEYLIST, \
                EM_REPLACESEL, 0, addr @szBuffer1
            inc      @dwIndex
        .endw
        invoke   RegCloseKey, @hKey
    .endif
ret

```

540

哭

例子程序中把注册表的各种操作分别写成子程序，用_RegQueryValue、_RegSetValue和_RegDelValue子程序来查询、设置和删除键值，用_RegCreateKey和_RegDelSubKey完成子键的创建和删除工作，并在WM_COMMAND消息中根据不同的按钮消息调用相应的子程序。为了让读者不经修改就可以将这些子程序用在其他程序中，将这些子程序放在一个单独的_Reg.asm文件中并在主程序中使用include语句包含进来，文件内容如下：

[illegible]

: Reg.asm 文件

[illegible]

: 查询键值

[illegible]

```

_RegQueryValue    proc    _lpzKey, _lpzValueName, \
                  _lpzValue, _lpdwSize, _lpdwType
                  local    @hKey, @dwReturn

                  mov     @dwReturn, -1
                  invoke   RegOpenKeyEx, HKEY_LOCAL_MACHINE, _lpzKey, NULL, \
                           KEY_QUERY_VALUE, addr @hKey
                  .if     eax == ERROR_SUCCESS
                     invoke   RegQueryValueEx, @hKey, _lpzValueName, \
                           NULL, _lpdwType, _lpzValue, _lpdwSize
                     mov     @dwReturn, eax
                     invoke   RegCloseKey, @hKey
                  .endif
                  mov     eax, @dwReturn
                  ret

```

RegQueryValue endp

[illegible]

: 设置键值

[illegible]

```

_RegSetValue      proc      _lpzKey, _lpzValueName, _lpzValue, \
                  _dwValueType, _dwSize
                  local     @hKey

                  invoke     RegCreateKey, HKEY_LOCAL_MACHINE, _lpzKey, addr @hKey
                  .if        eax == ERROR_SUCCESS
                      invoke   RegSetValueEx, @hKey, _lpzValueName, NULL, \
                                  _dwValueType, _lpzValue, _dwSize
                      invoke   RegCloseKey, @hKey
                  .endif
                  ret

```

RegSetValue endp

[illegible]

- 创建子键

注册表函数对注册表的操作是通过句柄来完成的，与文件操作一样，在对某个键下的子键或值项进行操作之前，需要先将这个键打开，然后使用键句柄来引用这个键，在操作完毕以后键句柄关闭。注册表的根键不需要打开，它们的句柄是固定不变的，要使用根键的时候只要

把这些句柄直接拿来用就是了，Windows.inc 中已经预定义了它们的数值：

HKEY_CLASSES_ROOT	equ 80000000h
HKEY_CURRENT_USER	equ 80000001h
HKEY_LOCAL_MACHINE	equ 80000002h
HKEY_USERS	equ 80000003h
HKEY_PERFORMANCE_DATA	equ 80000004h
HKEY_CURRENT_CONFIG	equ 80000005h
HKEY_DYN_DATA	equ 80000006h

在程序中可以随时将这些助记符当做句柄来引用对应的根键。在程序结束的时候，不需要关闭这些根键句柄。

打开子键使用 RegOpenKeyEx 函数，在 Win16 中还存在一个 RegOpenKey 函数，虽然在 Win32 中这个函数仍然存在，但这仅是为了兼容的目的而设置的。API 手册中推荐使用 RegOpenKeyEx 函数：

invoke	RegOpenKeyEx, hKey, lpSubKey, dwOptions, samDesired, phkResult
--------	--

函数的 hKey 参数指定父键句柄，lpSubKey 指向一个字符串，用来表示要打开的子键名称，在系统中一个子键的全称是以“根键\第 1 层子键\第 2 层子键\第 n 层子键”类型的字符串表示的，中间用“\”隔开，字符串的最后以 0 字符结束，这和目录名的表示方法是很像的。

既然子键的全称是这样表示的，那么要打开一个子键的时候，下面的两种表示方法有什么不同呢？

(1) 父键=HKEY_LOCAL_MACHINE, 子键=Software\RegTest\MySubkey
(2) 父键=HKEY_LOCAL_MACHINE\Software, 子键=RegTest\MySubkey

答案是：这两种表示方法是完全相同的。在使用 RegOpenKeyEx 函数打开子键的时候，既可以将 hKey 参数设置为 HKEY_LOCAL_MACHINE 根键的句柄，并将 lpSubKey 参数指向“Software\RegTest\MySubkey”字符串；也可以将 hKey 参数设置为“HKEY_LOCAL_MACHINE\Software”的句柄，将 lpSubKey 参数指向“RegTest\MySubkey”字符串，得到的结果是一样的。但是，使用第一种方法时，hKey 参数可以直接使用助记符 HKEY_LOCAL_MACHINE 来表示，因为根键的句柄是固定的，不需要打开；而使用第二种方法时，还需要先打开“HKEY_LOCAL_MACHINE\Software”键来获取它的句柄，所以具体使用哪种方法还要根据具体情况灵活选用。

函数的其他几个参数的含义如下。

- dwOptions 参数——系统保留参数，必须指定为 0。
- samDesired 参数——子键的打开方式，根据使用子键的方式，可以设置为下列取值的组合，只有指定了打开的方式，才能在打开子键后进行相应的操作：
 - KEY_ALL_ACCESS——允许所有的存取。
 - KEY_CREATE_LINK——允许建立符号列表。
 - KEY_CREATE_SUB_KEY——允许建立下一层子键。

- KEY_ENUMERATE_SUB_KEYS——允许枚举下一层子键。
- KEY_EXECUTE——允许读操作。
- KEY_QUERY_VALUE——允许查询键值数据。
- KEY_READ—KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS 和 KEY_NOTIFY 的组合。
- KEY_SET_VALUE——允许修改或创建键值数据。
- KEY_WRITE——KEY_SET_VALUE 和 KEY_CREATE_SUB_KEY 的组合。
- phkResult 参数——指向一个双字变量，函数在这里返回打开的子键句柄。

如果函数执行成功，返回值是 ERROR_SUCCESS，并且函数在 phkResult 参数指向的变量中返回子键句柄。

当不再需要继续使用键句柄的时候，可以使用 RegCloseKey 函数将它关闭：

invoke RegCloseKey, hKey

如果句柄被成功关闭，函数返回 ERROR_SUCCESS。

2. 创建和删除子键

创建一个子键可以使用 RegCreateKeyEx 函数：

invoke RegCreateKeyEx, hKey, lpSubKey, Reserved, lpClass, dwOptions, \
 samDesired, lpSecurityAttributes, phkResult, lpdwDisposition

函数中与 RegOpenKeyEx 函数中同名参数的含义和用法是相同的，hKey 也是用来指定父键句柄，lpSubKey 指向要创建的子键名称字符串，samDesired 参数指明子键建立后的操作方式，phkResult 指向用来返回键句柄的双字变量。

其余一些参数的含义如下：

- Reserved 参数——保留参数，必须设置为 0。
- lpClass 参数——为创建的子键定义一个类名，这个参数一般设置为 NULL。
- dwOptions 参数——创建子键时的选项，它可以是以下取值之一：
 - REG_OPTION_NON_VOLATILE——默认值，子键被创建到注册表文件中。
 - REG_OPTION_VOLATILE——创建易失性的子键，子键被保存在内存中，当系统重新启动的时候，子键消失。这个选项仅对 Windows NT 系统有效，在 9x 系统中被忽略。
- lpSecurityAttributes 参数——指向一个 SECURITY_ATTRIBUTES 结构，用来指定键句柄的继承性，如果句柄不需要被继承，可以使用 NULL。
- lpdwDisposition 参数——这个参数一般使用 NULL。

当需要创建的子键已经存在的时候，函数仅起到 RegOpenKeyEx 函数的作用；如果子键不存在，那么函数将创建子键。如果函数执行成功，返回值是 ERROR_SUCCESS。

如果要创建“HKEY_LOCAL_MACHINE\Key1\Key2\Key3”子键，既可以将 hKey 参数设置为 HKEY_LOCAL_MACHINE，将 lpSubKey 参数指向“Key1\Key2\Key3”字符串；也可以先打开“HKEY_LOCAL_MACHINE\Key1”键，将 hKey 设置为打开的键句柄，然后将 lpSubKey 参数指向“Key2\Key3”字符串，这与 RegOpenKeyEx 函数中的用法是类似的。在第二种用法中，打开父键的时候注意要指定 KEY_CREATE_SUB_KEY 方式。

当被创建子键的上层键不存在的时候，函数连同上层的子键一起创建。如上面的例子中，假如 Key2 也不存在，那么函数先在“HKEY_LOCAL_MACHINE\Key1”下创建 Key2，然后在 Key2 下继续创建 Key3。

_Reg.asm 文件中的大部分子程序首先用 RegOpenKeyEx 函数打开子键，以便进行下一步操作，但是保存键值用的_RegSetValue 子程序中使用的是 RegCreateKeyEx 函数，这样当子键已经存在的时候，函数仅打开它，如果子键不存在的话则创建子键。

删除子键使用 RegDeleteKey 函数：

invoke	RegDeleteKey, hKey, lpSubKey
--------	------------------------------

hKey 参数为父键句柄，lpSubKey 参数指向要删除的子键名称字符串。函数仅删除最后一层子键，以及下面的全部键值项。比如，在“HKEY_LOCAL_MACHINE\Key1\Key2\Key3”子键存在的情况下，当 hKey 指定为 HKEY_LOCAL_MACHINE，lpSubKey 指向“Key1\Key2\Key3”的时候，函数仅删除 Key3 子键，不会连同 Key2，Key1 全部删除。但如果 Key3 子键下有键值项的话，这些键值项会被一起删除。

如果要删除的子键下还存在下一层子键，比如，上例中的 Key3 子键下还存在 Key4 子键，那么对 Key3 子键进行删除时，Windows 9x 和 Windows NT 系统的做法是不同的：在 Windows 9x 中，Key3 子键本身、Key3 子键下所有的键值项和下层子键（包括上面举例的 Key4）会被全部删除；而在 Windows NT 中，只有在不存在下层子键的情况下删除才能成功，如果 Key3 子键下还存在 Key4 子键，那么对 Key3 子键的删除是不会成功的。



应用程序不能直接在 HKEY_LOCAL_MACHINE 根键下面创建和删除子键，只能在下一层由系统定义的子键下进行操作，如果要保存配置信息的话，用户应用程序一般在 HKEY_LOCAL_MACHINE\SOFTWARE 子键下再创建自己的子键，然后将键值项保存在自己的子键中。

15.3.3 管理键值

配置信息是存放在键值项中的，打开或者创建一个键的最终目的都是为了在键下面存取键值项，这就像磁盘上的目录是用来合理组织和管理文件用的，数据还是存放在文件中的。当使用打开或者创建键的函数得到键句柄后，就可以通过它来存取键值项了。

1. 设置键值项

在一个键下面设置和创建键值项使用 RegSetValueEx 函数：

invoke	RegSetValueEx, hKey, lpValueName, Reserved, dwType, lpData, cbData
--------	--

- hKey 参数指定一个键句柄，键值项将保存在这个键下，lpValueName 参数指向定义键值项名称的字符串。假如 lpValueName 参数指向一个空串或者设置为 NULL，并且设置的键值类型是 REG_SZ 的话，那么函数设置的是键的默认值（图 15.2 中所示的“默认”项）。
- Reserved 参数是保留的，必须设置为 0。
- dwType 参数指出了要设置的键值数据的类型，可以使用的类型如表 15.1 所示。
- lpData 参数是一个指针，指向包含键值数据的缓冲区，cbData 参数为要保存的数据长度。缓冲区中的数据格式，以及 cbData 参数指定的数据长度需要和 dwType 参数指出的键值类型相对应，比如，要设置 REG_SZ 类型的键值项，就要将 cbData 参数设置为字符串的长度+1（加上尾部的 0）；同样对于 REG_MULTI_SZ 类型的键值项来说，最后的两个 0 的长度都必须包括到 cbData 参数中；对于 REG_DWORD 类型的键值项，需要将双字数据放在缓冲区中并将 cbData 参数设置为 4（不像其他函数一样当参数是双字的时候一般将双字在参数中直接传递）。

当子键中的键值项不存在的时候，函数新建键值项；当键值项已经存在的时候，函数将新的键值数据写入。如果键值数据保存成功，函数返回 ERROR_SUCCESS。

虽然键值数据的最大长度没有规定，其大小仅受限于可用的内存大小，应用程序甚至可以使用 REG_BINARY 格式的键值项将整个文件都保存到注册表中，但在实际的使用中还是建议不要将大于 2 KB 的数据放到注册表中，因为这将影响注册表的使用效率。

要在一个键中创建或修改键值项，键的打开方式中必须包括 KEY_SET_VALUE 方式。

2. 查询键值数据

读取键值项中的数据或者查询键值项的属性使用 RegQueryValueEx 函数，用法如下：

invoke	RegQueryValueEx, hKey, lpValueName, lpReserved, \
	lpType, lpData, lpcbData

参数 hKey 和 lpValueName 用来指定要读取的键值项所处的子键句柄和键值项的名称，lpReserved 参数是保留参数，必须使用 0。lpData 参数指向一个缓冲区，用来接收返回的键值数据。

函数的其余几个参数使用时必须注意的是它们都是指向双字变量的指针，这一点和使用 RegSetValueEx 函数时是不同的：

- lpType 参数——函数在这个参数指向的双字变量中返回读取的键值类型，如果不需要返回键值项的类型，可以将这个参数设置为 NULL。
- lpcbData 参数——在调用的时候，程序必须在这个参数指向的双字变量中放置缓冲区的长度（并不是直接用 lpcbData 参数指出缓冲区长度）。当函数返回的时候，双字变量被函数改为返回到缓冲区中的数据实际长度。

当函数执行成功的时候，函数的返回值是 ERROR_SUCCESS。当程序指定的缓冲区长度不足以容纳返回的数据的时候，函数的返回值是 ERROR_MORE_DATA，这时 lpcbData 参数指向的双字变量中返回需要的长度。

如果仅需要查询键值长度而不需要返回实际的数据，可以将 lpData 参数设置为 NULL，但是 lpcbData 参数不能为 NULL，这时函数会在 lpcbData 参数指向的双字变量中返回键值数据的长度。如果仅想查询键值项的类型，也可以同时将 lpcbData 和 lpData 参数设置为 NULL。在这些情况下如果函数查询成功，返回值也是 ERROR_SUCCESS。

如果要在一个键中查询键值数据的话，键的打开方式中必须包括 KEY_QUERY_VALUE 方式。

3. 删除键值项

删除一个键值项的操作则比较简单，使用 RegDeleteValue 函数就可以了：

invoke	RegDeleteValue, hKey, lpValueName
--------	-----------------------------------

hKey 参数和 lpValueName 指定父键句柄和被删除键值项的名称。惟一需要注意的是父键句柄的打开方式必须包括 KEY_SET_VALUE。如果键值项被成功删除，则函数返回 ERROR_SUCCESS。

15.3.4 子键和键值的枚举

在实际的应用中往往需要对一个键下的子键或者键值项进行列表操作，就像在 DOS 系统下常用 Dir 命令一样，这就要用到子键和键值的枚举函数。在注册表函数中，枚举子键和枚举键值项使用的函数是不一样的，不像 FindFirstFile 等文件列表函数那样将文件连同子目录混在一起列出来。下面分别介绍这两种函数。

1. 枚举子键

例子程序中枚举子键和键值项的操作是在 _EnumKey 子程序中完成的，读者可以参考一下相应的代码，在这个子程序中，程序首先使用 RegEnumKeyEx 函数来枚举子键：

invoke	RegEnumKeyEx, hKey, dwIndex, lpName, lpcbName, lpReserved, \\ lpClass, lpcbClass, lpftLastWriteTime
--------	--

- hKey 参数指定被枚举的键句柄，dwIndex 参数指定需要返回信息的子键索引编号，lpName 指向一个缓冲区，函数在这里返回子键名称，lpClass 指向用于返回子键类名的缓冲区，lpftLastWriteTime 指向一个 FILETIME 结构，函数在这里返回子键上

一次被写入的时间。lpReserved 参数是保留参数，必须设置为 0。

要注意的是：lpcbName 和 lpcbClass 指向两个双字变量，调用函数前，这两个双字变量中必须放入 lpName 和 lpClass 指定的缓冲区的长度，当函数返回的时候，函数在里面返回实际返回到缓冲区中的字符串的长度。如果函数执行成功，返回值是 ERROR_SUCCESS。

因为 RegEnumKeyEx 函数每次返回一个子键的名称信息，所以要枚举全部子键的话，必须用循环多次调用这个函数，并且每次将 dwIndex 参数指定的子键索引号递增，当子键全部被枚举后，继续调用函数将得到一个 ERROR_NO_MORE_ITEMS 返回值，这时就可以结束循环了。下面是循环的典型写法：

```

        .data
dwIndex      dd      ?
dwSize       dd      ?
szBuffer db 256 dup (?)
        .code
        ... ..
        mov     dwIndex, 0
        .while  TRUE
            mov     dwSize, sizeof szBuffer
            invoke  RegEnumKeyEx, hKey, dwIndex, addr szBuffer, \
                addr dwSize, NULL, NULL, NULL, NULL
            .break .if eax == ERROR_NO_MORE_ITEMS
            ;处理获取的子键
            inc     dwIndex
        .endw

```

在循环开始前，程序初始化当做索引用的 dwIndex 变量，每次调用 RegEnumKeyEx 后将索引加 1，当检测到函数的返回值是 ERROR_NO_MORE_ITEMS 的时候，使用 .break 语句退出循环。程序不使用 .break .if eax != ERROR_SUCCESS 语句当做结束循环的条件是因为：当出现缓冲区不够长等意外情况时，函数的调用可能失败，但是这时子键可能还没有全部被枚举，所以只有判断返回值是 ERROR_NO_MORE_ITEMS 才能保证全部子键被枚举。

每次调用函数之前，程序必须重新将 dwSize 变量的值设置为 szBuffer 缓冲区的大小，这一点非常重要，笔者有很多次听朋友说他的程序只能枚举一部分子键，最后查出的原因就是忘了这一步。这是因为每次函数返回时，dwSize 中会变成返回的子键名称字符串的长度，如果不重新设置，下一次调用时函数就会将这个长度认为是缓存区的长度。随着循环的进行，这个值会变得越来越小。

当进行枚举子键操作时，父键的打开方式中必须包括 KEY_ENUMERATE_SUB_KEYS 方式（KEY_READ 方式中已经包括 KEY_ENUMERATE_SUB_KEYS）。

2. 枚举键值

RegEnumKeyEx 函数仅枚举一个键下面的全部子键，对键下面的键值项则不会去理会。如果要枚举一个键下面的键值项，那么必须使用 RegEnumValue 函数：

```

invoke  RegEnumValue, hKey, dwIndex, lpValueName, lpcbValueName, \
        lpReserved, lpType, lpData, lpcbData

```

函数的 `hKey`, `dwIndex` 和 `lpReserved` 参数的使用同 `RegEnumKeyEx` 函数中的同名参数。其余的一些参数中, `lpValueName` 和 `lpData` 参数指向两个缓存区, 函数在里面分别返回键值项的名称和数据。 `lpcbValueName` 和 `lpcbData` 参数指向两个双字变量, 调用函数前里面必须放入键值项名称缓冲区和键值数据缓冲区的长度, 函数返回后这两个变量的值被改为返回到缓冲区中的数据长度。 `lpType` 参数则指向一个用于返回键值数据类型的双字变量。

如果不需要返回键值数据, `lpData` 和 `lpcbData` 参数可以设置为 `NULL`, 如果不需要返回键值数据类型, `lpType` 参数也可以设置为 `NULL`。

要进行枚举键值项的操作, 父键的打开方式中必须包括 `KEY_QUERY_VALUE` 方式。

下面是一段典型的用于枚举键值项的循环代码:

```
.data
dwIndex      dd      ?
dwType       dd      ?
dwNameSize   dd      ?
szName       db      256 dup (?)
dwDataSize   dd      ?
szData       db      256 dup (?)
.code
...
mov          dwIndex, 0
.while      TRUE
    mov      dwNameSize, sizeof szName
    mov      dwDataSize, sizeof szData
    invoke   RegEnumValue, hKey, dwIndex, addr szName, \
              addr dwNameSize, NULL, addr dwType, \
              addr szData, addr dwDataSize
    .break   .if eax == ERROR_NO_MORE_ITEMS
    ;处理获取的键值项
    inc      dwIndex
.endw
```

这个循环的结构和使用 `RegEnumKeyEx` 函数的循环是大同小异的。同样要注意的是, 在循环中每次要重新设置 `dwNameSize` 和 `dwDataSize` 变量的值, 因为它们在 `RegEnumValue` 函数执行后也会被改写。

3. 查询键属性

在枚举子键和键值项的时候往往会遇到这样一个问题: 注册表函数对键值数据的长度并没有限制, 在预留缓冲区的时候如果申请太大的内存比较浪费, 申请太小的内存则无法枚举成功, 对于返回的子键名称和键值项名称也是如此。那么, 究竟该留多大的缓冲区呢? 其实在枚举之前可以先用 `RegQueryInfoKey` 函数查看一下键的统计信息。

`RegQueryInfoKey` 函数返回的信息有: 一个键下面子键的数量、键值项的数量、子键名称和键值名称字符串的最大长度及键值数据的最大长度等。根据这些信息, 就能方便地申请足够大的缓冲区来保证枚举成功。函数还能返回创建子键时指定的类名和最后一次写入子键的时间等信息。

RegQueryInfoKey 函数的用法如下：

invoke	RegQueryInfoKey, hKey, lpClass, lpcbClass, lpReserved, \
	lpcSubKeys, lpcbMaxSubKeyLen, lpcbMaxClassLen, \
	lpcValues, lpcbMaxValueNameLen, lpcbMaxValueLen, \
	lpcbSecurityDescriptor, lpftLastWriteTime

函数的参数比较多，但并不复杂，各参数的含义是：

- hKey——指定要获取信息的键句柄，键的打开方式中必须包括 KEY_QUERY_VALUE。
- lpClass——指向一个缓冲区，用来返回创建键时指定的 Class 字符串。
- lpcbClass——指向一个双字变量，调用函数时变量中必须放入 lpClass 指定的缓冲区的长度，函数返回时在这里放入返回到缓冲区中的字符串长度。
- lpReserved——保留参数，必须设置为 0。
- lpcSubKeys——指向一个双字，用来返回键中的子键数量。
- lpcbMaxSubKeyLen——指向一个双字，用来返回所有子键中最长的名称字符串长度，返回的长度不包括字符串结尾的 0 字符。
- lpcbMaxClassLen——指向一个双字，用来返回所有子键中最长的 Class 字符串长度，返回的长度不包括字符串结尾的 0 字符。
- lpcValues——指向一个双字，用来返回键下面的键值项数量。
- lpcbMaxValueNameLen——指向一个双字，用来返回所有键值项中最长的名称字符串长度，返回的长度不包括字符串结尾的 0 字符。
- lpcbMaxValueLen——指向一个双字，用来返回所有键值数据的最大长度。
- lpcbSecurityDescriptor——指向一个双字，用来返回安全描述符的长度。
- lpftLastWriteTime——指向一个 FILETIME 结构，用来返回最后一次修改键的时间。

可以看到，除 hKey 外其他的参数都是指针，指向用来返回数据的变量或结构，如果不需要返回某种信息的话，可以将对应的指针参数设置为 NULL。另外，所有返回的最长名称字符串长度中都不包括结尾的 0 字符。

除了前面介绍的函数外，系统中还存在一些不常用的注册表函数，比如，可以用 RegLoadKey 和 RegReplaceKey 函数从指定的文件中恢复注册表的子键信息，也可以通过 RegSaveKey 函数将键信息保存到指定的文件中。另外，可以通过 RegConnectRegistry 等函数操作远程注册表。对于这些函数，本节不再详细介绍。

15.3.5 注册表应用举例

曾经有一段时间，注册表修改和优化的工具层出不穷，如魔法兔子、侠客系统修改器和 Windows 优化大师等都是这方面的典型软件。看了前面的注册表函数后，读者现在一定知道，编写这些软件的大部分时间是花在界面设计和收集注册表的说明资料上，在这些工作的背后，使用的就是这么几个注册表函数。

本节用几个简单的例子来演示一些涉及注册表操作的常见应用，读者在收集到足够的注册表说明资料后，参考这些例子就可以写出类似的应用程序来。

1. 设置开机自动运行

Windows 在启动并执行登录操作后，会将 HKEY_LOCAL_MACHINE\ Software\ Microsoft\Windows\CurrentVersion\Run 子键下的所有键值项枚举一遍，并将所有 REG_SZ 类型的键值数据字符串当做一个文件名自动执行，所以在这个子键下设置一个键值项，让它的键值数据是某个文件名字符串（也可以是数据文件名，Windows 会自动打开关联的可执行文件），就可以让这个文件在 Windows 启动后自动运行。

由于 Windows 只关心键值数据，并不关心键值名称，所以在设置的时候只要保证键值名称是惟一的就可以了。下面的_SetAutoRun 子程序就可以用来将程序设置为自动运行：

```

                .const
szKeyAutoRun   db   'Software\Microsoft\Windows\CurrentVersion\Run',0
szValueAutoRun db   'AutoRun Test',0    ;在不同程序中使用修改此字符串！
                .code
                ... ..
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_SetAutoRun    proc    _dwFlag
                local   @szFileName[MAX_PATH]:byte

                .if     _dwFlag
                invoke   GetModuleFileName, NULL, addr @szFileName, MAX_PATH
                inc      eax
                invoke   _RegSetValue, addr szKeyAutoRun, addr szValueAutoRun, \
                        addr @szFileName, REG_SZ, eax
                .else
                invoke   _RegDelValue, addr szKeyAutoRun, addr szValueAutoRun
                .endif
                ret
_SetAutoRun    endp

```

当用 TRUE 参数调用_SetAutoRun 子程序时，程序将被设置为自动运行，这时子程序先用 GetModuleFileName 函数获取当前执行文件的文件名，然后用 .const 段中定义的 szValueAutoRun 字符串当做键值项的名称，用文件名当做键值数据在 Run 键下设置一个键值项。当用 FALSE 参数调用的时候，程序将取消自动运行，这时子程序仅简单地将前面设置的键值项删除而已。

代码中用到的几个子程序在 15.3.2 小节中列出的_Reg.asm 文件中。

2. 设置文件关联

如果将一个数据文件与一个可执行文件关联，那么就可以通过双击数据文件来直接执行可执行文件，比如，双击以 txt 为扩展名的文本文件，系统就会自动执行 Notepad.exe 文件来编辑它，这就是因为 txt 文件是与 Notepad.exe 文件关联的。

哭

第 16 章

WinSock 接口和网络编程

当今的时代是网络时代，网络给生活带来的影响超过了以往的任何事物，不管我们是用浏览器上网，是在打网络游戏，还是用 MSN、QQ 等即时通信软件和朋友聊天，网络的另一端实际上都是对应的网络应用程序在提供服务。

大多数的网络应用程序分为两部分：客户端和服务端。以浏览网页为例，IE 浏览器是客户端，Web 服务器是服务端；对于网络游戏，安装在我们的计算机上的游戏界面是客户端，而游戏服务商的服务器上运行的程序是服务端。本章将通过一些例子来说明如何用 Win32 汇编来编写网络应用程序的服务器端和客户端，由于篇幅有限，本章仅涉及如何使用 WinSock 接口编写网络应用程序，除此之外，重点介绍 TCP 网络应用程序架构的设计思想。

虽然不了解网络原理，但是并不妨碍写网络程序，可是了解了网络原理无疑会对正确设计程序架构带来很大的帮助，如果读者没有网络方面的基础知识，请首先花少量时间了解下面这些概念，这方面的内容可以参考网络上很常见的 CCNA 认证课程教材中的基础部分。

- 什么是 TCP/IP 协议？
- TCP 协议和 UDP 协议的特点是什么？
- IP 地址，地址掩码的含义是什么？
- 网络架构模型——开放系统互联模型（OSI 模型）的含义，该模型有哪些层次？平时常用的一些协议如 HTTP，FTP，TCP，UDP 和 ICMP 等协议分别位于哪个层次？
- TCP 栈的含义，数据在 TCP 栈中传递时是如何封装和解包的？
- 如果希望进一步了解细节，如各种协议的具体功能说明，数据包封装的详细定义等，可以阅读下面的资料：
- RFC 文档——TCP/IP 的技术标准是公开的，其技术标准都是以 RFC（Request for Comment）文档的形式出版的，读者可以很方便地从因特网中获取它们，获得任意一份 RFC 文档的最简单的方法是访问官方网站：

<http://www.rfc-editor.org/rfc.html>

- W. Richard Stevens 的遗作：3 卷中译本的《TCP/IP 详解》，它们是《卷 1：协议》、

《卷 2：实现》和《卷 3：TCP 事务协议、HTTP、NNTP 和 UNIX 域协议》，这 3 本书是网络编程资料中当之无愧的经典之作，书中详细地介绍了网络传输协议的实现细节，但内容并不涉及网络应用程序的编写。

16.1 Windows Socket 接口简介

Windows Socket 接口是 Windows 下网络编程的接口，在介绍 Windows Socket 接口之前，首先要简单介绍一下 TCP/IP 协议和描述网络系统架构的 OSI 模型，以及 TCP/IP 模型。

一般来说，网络系统的架构可以用开放系统互联模型（OSI 模型）来描述，OSI 模型分层的思想类似于 Windows 等操作系统的分层，在 Windows 下，应用程序位于最高层，应用程序通过 API 调用位于中间层次的系统子程序，系统子程序再调用驱动程序，驱动程序最终操作计算机的硬件，各层次之间的隔离有利于层次间的分工协作，只要每个层次都严格遵守边界协定，那么它对于其他层次来说就可以看成是一个“黑匣子”，结果就是开发人员能够致力于本层次的开发和提高，而不必担心能否和其他层次合作。与之类似，OSI 模型的体系结构分为 7 层，其中网络应用程序位于最高层，通过多个层次最终控制网络硬件所在的物理层来收发数据包。

TCP/IP 是 Transmission Control Protocol/Internet Protocol（传输控制协议/网际协议）的缩写，它最初是在 20 世纪 70 年代初期由美国国防部出资为 ARPA（美国高级研究项目局）开发的，经过了多年以后，以 TCP/IP 协议为基础构建的 ARPA 网逐步演变成了今天的 Internet。TCP/IP 网络的架构可以用 TCP/IP 模型来描述，这个模型和 OSI 模型极为相似，但是它将层次的划分减少到了 4 层，其每一层在功能上和 OSI 模型的一层或多层相对应，两种模型从工作原理上看并没有本质的区别。

图 16.1 描述了 OSI 模型和 TCP/IP 模型各层次之间的对应关系，并列举了部分在各层次上工作的网络协议，读者可以在其中看到很多熟悉的名词，如 Telnet，HTTP，TCP 和 UDP 等。

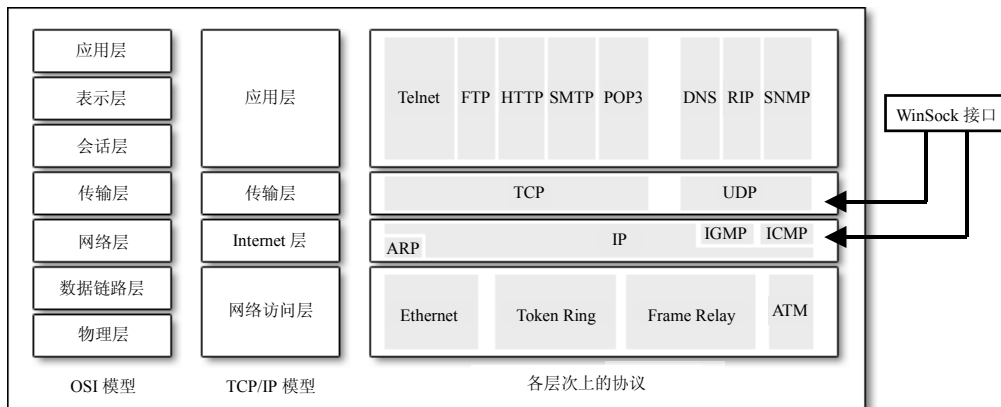


图 16.1 OSI 模型、TCP/IP 模型的结构和 WinSock 接口的关系

TCP/IP 协议的核心协议运行于传输层和 Internet 层上，主要包括 TCP，UDP 和 IP 协议，其中 TCP 协议和 UDP 协议是以 IP 协议为基础而封装的，这两种协议提供了不同方式的数据通

信服务。在后面的 16.2.3 节中，会对两种协议的区别做详细的介绍。

如果说 IP 协议是道路，那么下一层网络访问层的各种协议就相当于不同的铺路材料，而上一层的 TCP 和 UDP 协议就相当于路上跑的不同类型的车辆；再上层应用层的各种协议就相当于车上运送的丰富多彩的货物，它们都以 TCP 和 UDP 协议为载体来完成。比如，HTTP 协议使用 TCP 协议传输网页，POP3 协议使用 TCP 协议传输邮件，而 DNS 协议使用 UDP 协议来传输域名和 IP 地址的翻译信息。

Microsoft 为 Win32 环境下的网络编程提供了 Windows Sockets 接口(简称 WinSock 接口)，正如 Windows 下的各种接口都是以 API 的形式出现的一样(如 GDI)，WinSock 也是以一组 API 的方式提供的，从 1991 年推出 1.0 版开始，经过不断地完善后，WinSock 接口现在已成为 Windows 下网络编程的标准。

由于网络协议最早是在 UNIX 操作系统上实现的，所以从可移植性考虑，WinSock 接口以 BSD UNIX 操作系统中流行的 Socket 接口为范例定义，它包含了 UNIX Sockets 接口中一系列的同名函数，但是 Windows 系统的运行方式和 UNIX 系统有显著的不同，为了与 Windows 系统相适应，WinSock 接口在移植这些函数的同时对它们进行了改造，并针对 Windows 的消息驱动机制定义了一部分新的函数，所以 WinSock 接口函数实际上是 UNIX Sockets 接口函数的超集。

如图 16.1 所示，WinSock 接口提供的函数位于 TCP/IP 模型中的传输层和 Internet 层上面，也就是说，我们可以利用接口函数编写使用 TCP 和 UDP 协议的程序，也可以编写直接使用 IP 协议的程序，如使用 ICMP 协议完成 Ping 的功能，但 WinSock 接口并没有对应用层上的各种协议提供支持，所以应用程序无法通过 WinSock 接口提供的函数来实现 HTTP、FTP 与 Telnet 等协议，应用程序必须用另外的接口或者自己编程来实现这些高层协议。

与 GDI 等接口类似，WinSock 接口也是通过几个动态链接库来提供的，这些动态链接库从版本上分有 1.1 版本和 2.0 版本两种，从位数分可以分为 16 位版本和 32 位版本，如图 16.2 所示，WinSock 接口函数的代码主要包括在 WS2_32.dll 库文件中，这个库文件提供了对 2.0 版本 WinSock 接口的支持。在早期的 Windows 系统中，16 位和 32 位的 1.1 版本的文件名分别是 WinSock.dll 和 Wsock32.dll，为了给使用这些库文件的程序提供兼容性支持，系统中仍然存在这两个文件，只不过现在这两个文件中也是间接调用了 WS2_32.dll 文件而已。

为了使用 WinSock 接口，需要在源程序中包含对应的 inc 和 lib 文件，如果使用 2.0 版本，在源程序中必须包括 WS2_32.inc 和 WS2_32.lib 文件，如果要使用的是 1.1 版本的 WinSock 函数，那么既可以使用上面两个文件，也可以使用 Wsock32.inc 和 Wsock32.lib 文件。

在源文件中包含了对应的 inc 文件和 lib 文件后，在使用其他 WinSock 函数之前，必须首先使用 WSStartup 函数来装入并初始化动态链接库，否则对其他任何 WinSock 函数的调用都不会成功，WSStartup 函数只需要在程序开始的时候调用一次：

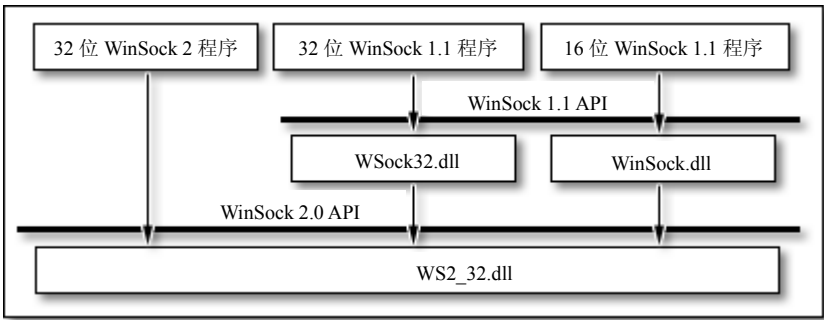


图 16.2 WinSock 接口使用的动态链接库

```
invoke    WSAStartup, wVersionRequested, lpWSAData
.if      eax
        ;无法初始化 WinSock 库
.endif
```

wVersionRequested 是一个 16 位的参数，用来指定动态链接库将支持哪个版本的 WinSock 函数，其中的低 8 位指定主版本号，高 8 位用来指定副版本号，假如要使用 1.1 版的，可以在这里使用 0101h，假如需要使用 2.0 版本函数，则可以将参数指定为 0002h。

lpWSAData 参数指向一个 WSADATA 结构，用来返回动态链接库的详细信息，结构的定义为：

```
WSADATA STRUCT
    wVersion      WORD      ?           ;库文件建议应用程序使用的版本
    wHighVersion  WORD      ?           ;库文件支持的最高 WinSock 版本
    szDescription BYTE  WSADESCRIPTION_LEN + 1 dup (?) ;库描述字符串
    szSystemStatus BYTE  WSASYS_STATUS_LEN + 1 dup (?) ;系统状态字符串
    iMaxSockets  WORD      ?           ;同时支持的最大套接字数量
    iMaxUdpDg    WORD      ?           ;2.0 版中已废弃的字段
    lpVendorInfo  DWORD    ?           ;2.0 版中已废弃的字段
WSADATA ENDS
```

szDescription 字段中返回的字符串一般是“WinSock 2.0”之类的库描述串，szSystemStatus 字段中返回的是类似于“Running”一类的运行状态字符串。如果库装入成功，函数将返回 0，否则将返回下面的出错代码：

- WSASYSNOTREADY——网络子系统未准备好。
- WSAVERNOTSUPPORTED——不支持指定的版本。
- WSAEINPROGRESS——另一个阻塞方式的 WinSock 1.1 操作正在进行中。
- WSAEPROCLIM——WinSock 接口已经到达了所支持的最大任务数。
- WSAEFAULT——输入参数 lpWSAData 指定的指针无效。

不需要再使用 WinSock 函数时，比如，在程序退出前，必须使用 WSACleanup 函数将库释放：

```
invoke    WSACleanup
```

WSACleanup 函数没有输入参数, 它将释放动态链接库并自动释放所有被创建的套接字等资源。如果函数执行成功将返回 0, 否则将返回 SOCKET_ERROR。



WinSock 函数中有些参数是 16 位的, 如 WSAStartup 函数中的 wVersionRequested 参数, 但是在传递这些参数时, 并不是堆栈中压入一个字 (word), 而是将它扩展到 32 位的双字 (dword) 以后再压入这个双字, 所以在源程序的 invoke 语句中并不需要特殊的处理。

16.2 Windows Socket 接口的使用

本节将介绍一些常用的 WinSock 函数, 但是在正式开始介绍前, 读者需要了解和这些函数密切相关的 IP 地址、端口和网络字节顺序等的重要概念。

16.2.1 IP 地址的转换

1. 什么是 IP 地址和端口

要在计算机之间使用 TCP/IP 协议进行通信, 这些计算机必须能够互相寻址, TCP/IP 协议使用 IP 地址寻址方式, 所以, 对于一个 IP 网络来说, 网络上的每个设备必须具有惟一的 IP 地址, 否则将无法正确地进行寻址, 同理, 要让整个 Internet 范围内的计算机都能够互相通信, Internet 上的所有设备也必须具有惟一的 IP 地址。

IP 地址是一个 32 位的二进制数, 为了用人们熟悉的十进制数表示, 通常将它分为 4 个 8 位的二进制数, 每个 8 位二进制数被转换成十进制, 中间用小数点隔开, 就得到了 IP 地址的十进制字符串格式, 图

<u>11000000</u>	<u>10101000</u>	<u>00000001</u>	<u>01100100</u>			
192	.	168	.	1	.	100

图 16.3 IP 地址的两种表示格式

16.3 中的 IP 地址 11000000 10101000 00000001 01100100, 以十进制方式表示就是 192.168.1.100。

既然计算机是以 IP 地址来辨别的, 如果一台计算机上运行了多个服务器程序怎么办? 比如, 既运行了 Web 服务, 又运行了 FTP 服务, 这两个服务都使用同样的 IP 地址, 那么数据到达这个 IP 地址后, 计算机如何分辨数据发往哪个服务的呢? 这就涉及协议复用的问题, 如果协议不能复用的话, 那么它就只能为一个进程服务。

为此, TCP 和 UDP 协议中有端口的概念, 使用这两种协议通信的时候, IP 地址和端口号结合起来才能惟一确定一个目标地址, 这样这两种协议就提供了同时为不同进程提供通信的能力, 在上面的例子中, Web 服务使用 80 号端口, FTP 服务使用 21 号端口, 客户端指定和 21 号端口连接, 系统就知道是和 FTP 服务进行通信。TCP/IP 模型其他层次的其他协议中并没有端口的概念, 如 ICMP 协议等, 这些协议就只能寻址到 IP 地址为止。

端口号用一个 16 位的整数来表示, 所以从理论上讲, 可以同时有 65 536 个服务使用同一 IP 地址进行通信, 由于传输层的 TCP 协议和 UDP 协议是两个完全独立的模块, 两者的工作是互不相干的, 所以 TCP 和 UDP 各自的端口号也相互独立, 一个进程使用 TCP 协议的某个端口号并

不影响另一进程使用 UDP 协议的同名端口号，但是同一协议的同端口号无法被两个进程同时使用。

大部分应用层上的协议都定义了自己使用的默认端口号（但这并不意味着必须强制使用这个端口号），另外，一些服务程序（如 SQL Server，Oracle 数据库与 Windows 的终端服务等）也使用固定的端口号。表 16.1 中列出了一些常用协议和服务程序使用的端口号，详尽的已分配端口号列表可以参考 RFC 1700 文档。

表 16.1 常用协议和应用程序使用的默认端口号

协议或应用程序	TCP 端口号	UDP 端口号
FTP	21	
Telnet	23	
SMTP	25	
HTTP	80	
POP3	110	
DNS 查询		53
TFTP 协议		69
NetBIOS 名字服务		137
NetBIOS 数据包服务		138
SQL Server 数据库	139、1433	
Oracle 数据库	1521	

由于在使用 TCP 和 UDP 协议进行通信时，必须同时指定 IP 地址和端口号才能完整地标识一个通信地址，所以在编程中通常将这两个参数一起定义在一个 `sockaddr_in` 结构中来使用，`sockaddr_in` 结构是 WinSock 接口中最常用的结构之一，它的定义是：

<code>sockaddr_in</code>	STRUCT	
<code>sin_family</code>	WORD	? ;地址格式
<code>sin_port</code>	WORD	? ;端口号（使用网络字节顺序）
<code>sin_addr</code>	<code>in_addr</code>	<> ;IP 地址（使用网络字节顺序）
<code>sin_zero</code>	BYTE	8 dup (?) ;空字节
<code>sockaddr_in</code>	ENDS	

结构中的 `sin_family` 字段用来指定地址格式，在不同操作系统下，取值可以指定为 `AF_UNSPEC`，`AF_UNIX` 或 `AF_OSI` 等不同的值，但是在 WinSock 中只能使用 `AF_INET`（也可以使用 `PF_INET`，在 `Windows.inc` 中 `PF_INET` 被定义为与 `AF_INET` 等效）。`sin_port` 字段和 `sin_addr` 字段则分别指定端口号和 IP 地址，其中 `sin_addr` 字段是个 `in_addr` 结构，这个结构实际上被定义为一个 `dword`。

`sockaddr_in` 结构看起来没什么特别的，但是数据放入 `sin_port` 字段和 `sin_addr` 字段时必须经过特殊处理，因为系统要求这两个字段的数据是按照网络字节顺序排列的。

2. 网络字节顺序

什么是网络字节顺序呢？这首先要从 CPU 对字节顺序的处理方式谈起。

CPU 对字节顺序的处理方式有两种：大尾方式 (big Endian) 和小尾方式 (little Endian)。在大尾方式中，数据的高字节被放置在连续存储区的首位，比如一个 32 位的十六进制数 12345678h 在内存中的存放方式是 12h, 34h, 56h, 78h，同样，IP 地址 192.168.0.1 在内存中的存放方式是 192, 168, 0, 1；而在小尾方式中，数据的低字节被放置在连续存储区的首位，上面的数据在内存中的存放方式变为 78h, 56h, 34h, 12h 及 1, 0, 168, 192。Intel 80x86 系列处理器和 DEC VAX 处理器使用的是小尾方式（所以我们常常看到内存中的多字节数是倒过来放置的），而 Motorola 的 680x0 和其他大部分的 RISC 芯片都使用大尾方式。

大尾和小尾方式各有好处，不同的处理器采用不同的方式本身无可厚非，但是要在它们之间进行通信的话就必须选定其中一种方式当做标准，否则会造成混淆，比如，某个采用 Intel CPU 的计算机要向某个采用 Motorola CPU 的计算机的 0100h 号端口发送数据，它按照自己的字节处理顺序在 TCP 数据包首部填入代表端口号的数据 00h, 01h（小尾方式下的 0100h），而接收方收到后却按照自己的方式理解为 0001h 端口，那就成问题了。

TCP/IP 协议是一组开放的协议，它被设计用来在不同的计算机平台之间进行通信，所以在协议的实现细节中不能包含与特定平台相关的内容，凡是与平台相关的都需要转换为规定的格式，其中最主要的就是对字节顺序的处理。

TCP/IP 协议统一规定使用大尾方式传输数据（也称 Internet 顺序），非常遗憾的是，这与 Intel 80x86 系列处理器使用的方式不同，所以在 80x86 平台下的 WinSock 编程中，当使用**需要在协议中使用的参数**时，必须首先将它们转换为 Internet 顺序。所以在填写 sockaddr_in 结构的 sin_port 字段和 sin_addr 字段时，必须首先进行转换。比如使用端口号 9 999，转换成十六进制是 270Fh，那么放入 sin_port 字段的数值就应该是转换以后的 0F27h，这样放到内存中后的排列顺序就刚好是 27h, 0Fh。

读者可能会问：为什么 sin_family 字段就不需要转换呢？这是因为 IP 地址和端口参数最终是被封装到 IP 数据包中通过网络传输到另一台计算机上去的，因此它们必须是网络字节顺序。但是 sin_family 字段是用来告诉 WinSocket 接口，结构中包含的地址类型是什么，并不需要发送到网络上，所以使用本机字节顺序。这就是前面说的“需要在协议中使用的参数”才需要转换的原因。

3. 字节顺序转换函数

为了方便进行字节顺序转换，WinSock 接口提供了一系列的函数。

htons 函数完成的功能是“Host to Network Short”，即将 16 位的以当前主机字节顺序排列的数据转换成按网络顺序排列的数据：

invoke	htons, hostshort
mov	netshort, ax

函数的输入值是按主机字节顺序排列的 16 位数据（当然调用时需要扩展到 32 位以便当做参数输入），返回值的低 16 位是转换后的按网络字节顺序排列的数据。

htonl 函数完成的功能是“Host to Network Long”，即将 32 位的以当前主机字节顺序排列的数据转换成按网络顺序排列的数据：

invoke	htonl, hostlong
mov	netlong, eax

ntohs 函数完成的功能是“Network to Host Short”，即将 16 位的按网络顺序排列的数据转换成以当前主机字节顺序排列的数据（输入参数同样需要首先被扩展到 32 位）：

invoke	ntohs, netshort
mov	hostshort, ax

ntohl 函数则完成“Network to Host Long”功能，即将 32 位的按网络顺序排列的数据转换成以当前主机字节顺序排列的数据：

invoke	ntohl, netlong
mov	hostlong, eax

细心的读者可能已经发现，函数名实际上是 n 和 h 的组合（分别代表 network 和 host），中间加上一个“to”，并且分 l 和 s（long 和 short）后缀而已。这些函数用于转换 IP 地址时，由于 IP 地址是 32 位的，需要使用 long 版本，转换端口时，由于端口是 16 位的，使用的是 short 版本。

一般来说，当涉及当前主机字节顺序和网络顺序数据的转换时，不管当前主机使用的字节顺序是否和网络顺序相同，最好都进行一次转换，而且转换时必须使用这些转换函数而不是自定义的函数，这样程序可以在不同的主机上进行移植。

比如，现在使用的是 Intel 80x86 平台，它的主机字节顺序和网络字节顺序是不一样的，如果使用自定义的函数将字节顺序反过来，万一 Windows 移植到和网络字节顺序一致的硬件平台上，转换的结果就不对了。反之，在运行于 Motorola 平台上的 UNIX 系统中，CPU 字节顺序和网络字节顺序是相同的，调用这些转换函数的话，函数的返回值和输入值不会有什么不同，但程序因此取消了字节顺序转换这一步骤的话，那么程序移植到运行于 Intel 平台上的 Linux 系统中时就会出错，所以不管主机的字节顺序是否和网络字节顺序相同，应该总是使用转换函数进行字节顺序转换。

4. 其他一些转换函数

除了字节顺序转换函数，WinSock 接口还提供了其他一些常用的转换函数，比如，将 32 位的 IP 地址和“aa.bb.cc.dd”类型的十进制 IP 地址字符串互相转换，这些函数可以为我们带来很多方便。

inet_addr 函数和 inet_ntoa 可以在 IP 地址和字符串之间进行转换。

inet_addr 函数将一个由小数点分隔的十进制 IP 地址字符串转换成由 32 位二进制数表示的 IP 地址（网络字节顺序）：

invoke	inet_addr, lpString
.if	eax != INADDR_NONE
mov	dwIP, eax
.endif	

lpString 参数指向“aa.bb.cc.dd”类型的 IP 地址字符串。如果转换成功，函数将返回已

经按网络字节顺序排列的 32 位 IP 地址，否则返回 INADDR_NONE。

inet_ntoa 则是 inet_addr 函数的逆函数，它将一个网络字节顺序的 32 位 IP 地址转换成字符串：

```
invoke    inet_ntoa, in
.if      eax
mov      lpsz, eax
.endif
```

参数 in 是需要转换的 32 位 IP 地址，如果转换失败函数返回 NULL，转换成功的话函数返回一个指针，指向转换后的 IP 地址字符串。这个字符串位于 WinSock 接口的内部缓冲区中，这是一个静态缓冲区，也就是说每次调用都使用同一个缓冲区，所以需要再次调用该函数前，必须将字符串拷贝到程序自己定义的缓冲区中，否则根据前面保存的指针来访问，得到的总是最后一次调用的结果。

一般来说，使用这些转换函数填写 sockaddr_in 结构的方式如下：

```
invoke    inet_addr, addr szIpAddress ; szIpAddress = 'aa.bb.cc.dd'
.if      eax == INADDR_NONE
jmp      Error
.endif
mov      @stSin.sin_addr, eax
mov      @stSin.sin_family, AF_INET
invoke    htons, 12345 ;假设端口号是 12345
mov      @stSin.sin_port, ax
```

例子中的@stSin 定义成 sockaddr_in 结构，使用的端口号是 12345，请注意 inet_addr 函数返回的整数类型的 IP 地址已经是网络字节顺序的，所以不需要调用 htonl 函数进行转换就可直接放入 sin_addr 字段，而端口号 12345 必须先使用 htons 函数进行转换后才能放入 sin_port 字段。

16.2.2 套接字

了解了 IP 地址、网络字节顺序等概念后，我们还无法开始写网络程序，还需要来了解一个新的概念。大家都知道，网络应用程序一般也被称为“socket 程序”，UNIX 和 Windows 操作系统中的网络编程接口都被称为 socket 接口，那么什么是 socket 呢？

1. 什么是套接字 (socket)

为了使用 WinSock 接口进行通信，首先必须建立一个用来通信的对象，这个对象就称为套接字 (socket)，套接字的定义是“通信的一端”，在通信的另一端必定有另一个套接字与之相对应以便互相传递数据。仅从编程的角度来看，套接字就是一个句柄而已，但 socket 这个称呼比句柄更贴切，因为 socket 的英文含义是插座、插孔或者接口，表达的含义就是通信时两端的对象必须一一对应接在一起才能工作。

套接字的种类有很多种，最主要的两种是流套接字 (stream socket) 和数据报套接字 (datagram socket)。由于流套接字使用传输层的 TCP 协议进行通信，所以它具有 TCP 协议

所拥有的各种特征，比如，它是面向连接的、稳定的，以及数据包是顺序发送的；而数据报套接字使用 UDP 协议进行通信，所以它的特征来自于 UDP 协议，如数据包可能丢失，可能重复，以及可能不按顺序到达等（一般将这两种套接字更直观地称为 TCP 套接字和 UDP 套接字，本书后面的内容也使用这种直观的称呼方式）。

另外，也存在其他一些不常用的套接字类型，如原始套接字（raw socket）、可靠信息分递套接字（rdm socket）和连续分包套接字（seqpacket socket）等，其中值得一提的是原始套接字，由于使用这种套接字可以直接在 Internet 层上处理 IP 数据包的首部，所以可以用它来实现各种特殊的功能，如伪造发送者地址等。

使用 WinSock 接口进行通信的时候，第一个步骤就是建立一个套接字，由于套接字在创建时就需要指定种类，所以一旦套接字被创建，那么使用该套接字进行通信时使用的协议也就已经基本确定了。读者不可能创建一个 TCP 套接字后再希望通过它发送 UDP 数据包。



有一部分介绍 WinSock 的书籍中讲到：“套接字的种类有两种——流套接字和数据报套接字”，这种说法是不确切的，因为正如上面介绍的，实际上还存在其他类型的套接字。

2. 套接字的创建和关闭

要开始通信之前，必须首先创建一个套接字，创建套接字使用 socket 函数：

```
invoke    socket, af, type, protocol
.if      eax != INVALID_SOCKET
mov      hSocket, eax
.endif
```

函数的第一个参数 af 用来指定套接字使用的地址格式。这个参数和 sockaddr_in 结构中 sin_family 字段的定义是一样的，惟一可以使用的值是 AF_INET。

第二个参数 type 用来指定套接字的类型。正如前面介绍的，套接字有流套接字、数据报套接字和原始套接字等，下面是最常用的几种套接字类型定义：

- SOCK_STREAM——流套接字，使用 TCP 协议提供有连接的和可靠的传输。
- SOCK_DGRAM——数据报套接字，使用 UDP 协议提供无连接的和不可靠的传输。
- SOCK_RAW——原始套接字，WinSock 接口并不使用某种特定的协议去封装它，而是由程序自行处理数据包，以及协议首部。

protocol 参数配合 type 参数使用，用来指定使用的协议类型，当 type 参数指定为 SOCK_STREAM 或者 SOCK_DGRAM 的时候，系统已经明确使用 TCP 和 UDP 协议来工作，所以这时这个参数并没有什么意义，可以指定为 0，但是当 type 参数指定为 SOCK_RAW 类型的时候，使用 protocol 参数可以更详细地指定原始套接字的工作方式。

当 type 参数指定为 SOCK_RAW 类型时，可以将 protocol 参数指定为以下的数值：

- IPPROTO_IP, IPPROTO_ICMP, IPPROTO_TCP 和 IPPROTO_UDP——分别指定使用 IP,

ICMP, TCP 和 UDP 协议, 这时系统会自动为数据加上 IP 首部, 并且将 IP 首部中的上层协议字段设置为指定的这些协议名称。但是使用这个套接字接收数据时, 系统却不会将 IP 首部自动去除, 需要程序自行分析处理 (如果在以后将套接字的属性设置上 IP_HDRINCL 选项的话, 那么发送时系统将不自动添加 IP 首部, 这时需要自己封装数据包)。

- IPPROTO_RAW——系统将数据包直接送到网络访问层发送, 程序需要自己添加 IP 首部, 以及其他协议首部, 并且需要自己计算和填充协议首部中的校验和字段。当使用 IPPROTO_RAW 协议类型的原始套接字时, 这个套接字只能用来发送数据包而无法接收数据包。这是因为所有的 IP 包都是先递交给系统核心, 然后再传输到用户程序, 当发送这样一个原始数据包的时候, 核心并不知道, 也没有这个数据被发送或者连接建立的记录, 因此当远端主机回应的时候, 系统核心就把这些包给丢弃而不是送到应用程序中。

当套接字被成功创建的时候, 函数将返回一个套接字句柄, 否则函数将返回 INVALID_SOCKET, 这时可以继续调用 WSAGetLastError 函数获取更详细的出错信息。

当不再需要使用套接字的时候, 需要使用 closesocket 函数将它关闭:

```
invoke    closesocket, s
```

参数 s 就是创建套接字时返回的套接字句柄。

对于还处于连接中的 TCP 套接字来说, 关闭了套接字, 系统会自动断开对应的连接。



当出错的时候, 大部分 WinSock 函数的返回值是 INVALID_SOCKET 或者 SOCKET_ERROR, 如果需要进一步的出错代码, 必须马上调用 WSAGetLastError 来获取, 在以后提及“出错代码”时, 指的就是这样得到的出错代码而不是函数的返回值。但惟一的例外是 WSStartup, 它出错时会直接返回出错代码, 因为这时 WinSock 库尚未装入, WSAGetLastError 函数还无法工作。

3. 套接字的工作模式

WinSock 套接字的使用分为两种模式: 阻塞模式和非阻塞模式。阻塞模式也称为同步模式, 在这种模式下, WinSock 函数会等待操作完成后才返回。比如, 使用 recv 函数接收数据时, 如果函数被调用时没有数据可收, 那么函数不会返回, 直到收到一些数据为止; 再比如使用 send 函数发送 n 字节的数据时, 在全部 n 字节的数据发送完之前, 函数不会返回, 所以这种模式下, 调用 WinSock 函数的线程有可能处于等待状态。

在 BSD UNIX 中, 套接字是以阻塞模式执行的, 这对以命令行方式执行的 UNIX 程序来说并不是问题, 但在 Windows 系统中运行时, 以阻塞模式工作的 socket 函数 WinSock 函数必须放在单独的线程中, 如果放在主线程中工作, 那么主线程可能被阻塞而导致界面无法响应。为了适应 Windows 下的消息驱动体系, WinSock 接口可以让套接字工作在非阻塞模式下, 非阻塞模式又称异步模式, 在这种模式下, 同样是前面所述的情况, recv 或 send 函数执行后会立即返

回，等有数据到达或者链路空闲可以继续发送数据时，WinSock 接口会通过某种形式（如窗口消息）通知应用程序，显然这种方式比较适合于 Windows 下的消息驱动体系。

具体编程的时候，究竟采用哪种模式并没有定论，每种模式都有自己的优缺点，在后面的例子中，会有不同的例子对两种模式的工作方式进行对比。当一个套接字被创建的时候，它默认工作在阻塞模式下，设置成非阻塞模式的方法，以及在该模式下的消息驱动机制将在 16.3.4 节中做详细介绍。

16.2.3 网络应用程序的一般工作流程

到这里，我们离网络编程又近了一步，但是我们还是应该首先了解一下网络应用程序的常用架构，以及 TCP、UDP 协议的详细特征，并初步了解一下后面将要介绍的函数都会用在架构中的哪个部分，这样在遇到具体的函数时可以做到心中有数。

从程序的结构及用途来看，大多数的网络应用程序分为两部分：客户端和服务端。从使用的传输协议上看，占绝对多数的应用程序使用 TCP 和 UDP 两种协议，只有少量程序使用其他协议（如大家熟悉的 Ping 程序使用 ICMP 协议）。虽然在应用层上的程序使用了例如 HTTP, FTP, POP3 以及 DNS 等很多种协议，但这些协议本质上还是对 TCP 和 UDP 协议的封装。

1. TCP 协议的特征和 TCP 程序的工作流程

TCP 协议是一种传输层上的协议，它提供一种面向连接的、可靠的字节流服务。

面向连接的含义是：两个 TCP 套接字在开始传输数据之前必须先建立一个连接，也就是说由一方首先向另一方发送请求信息，双方互相确认以后才能开始通信。这一过程与打电话很相似，一方先拨号振铃，等待对方摘机后才开始对话，如果对端没有程序响应，那就像没有人接电话一样，通信是无法开始的。另外，面向连接意味着 TCP 协议不能用于广播，即一方不能用一个套接字同时向多方发送数据。

字节流服务的含义是：TCP 连接上传输的数据是流方式的，也即没有边界的，假设发送方分三次分别发送了 100、150、200 字节的数据包，接收方看到的是一段数据流，无法得知发送方是如何分割发送的，接收方既可以一次接收 450 字节来将全部数据收完，也可以一次接收 45 字节分十次将数据收完。

TCP 通信是可靠的，它采用超时及重传机制来保证不丢失数据。当一个 TCP 发送一个数据包后，它将启动一个定时器，等待对端确认收到这个包，如果在指定的时间内没有得到确认，将重发这个数据包。而接收数据包的时候，它将发送一个确认，如果检测发现数据包的校验和有错，TCP 协议丢弃这个数据包并且不发送确认，那么对端会因为超时而重新发送这个数据包。

数据包在传输的时候会通过多个路由器，不同的数据包到达终点前经过的路由器可能是不同的，因此所花的时间也是不同的，这就可能造成后发的数据先到的情况，TCP 协议在 TCP 首部保存数据包序号，如果有必要，它将对收到的数据进行重新排序，并将收到的数据以正确的顺序交给应用层。

接收方收到的 IP 报文段也可能重复，原因之一是在发送和确认之间还有个时差，发送方

可能因为接收方的确认信息还在路上就发生超时而重发数据，这时接收方收到的数据包就会发生重复，对于这种情况，接收方会丢弃重复的数据。

TCP 协议还提供流量控制机制，发送方可以根据接收方应答的时间和速率适当调整自己的数据发送速度，这样可以防止速度较快的主机使速度较慢主机的接收缓冲区溢出。

TCP 协议的工作过程也和打电话的过程很相似。当一方滔滔不绝地讲话时，需要另一方偶尔回复“是的”、“嗯”之类的短语来确认，如果有一段时间没有听到对方的简短确认，发言方就会停下来问一句“你在听吗”。当另一方没有听清楚某句话的时候，他会说：“你刚才说什么，我没有听清楚”，这样发言方会复述前面的句子。如果一方讲得太快，另一方会要求他适当放慢速度。TCP 协议实现的机制就与此类似。

从 TCP 协议的特征可以看出通信双方的工作方式必然是不同的，这种工作方式的不同可以用客户机/服务器模型 (Client/Server 或 C/S) 来描述，通信的发起端被称为客户机 (Client，也称为工作站端或客户端)，通信的等待方被称为服务器 (Server，也称为服务器端)，图 16.4 显示了两者的工作方式的不同，图中的括号内显示了各步骤使用的 WinSock 函数，这些函数的用法在后面会有更详细的介绍。

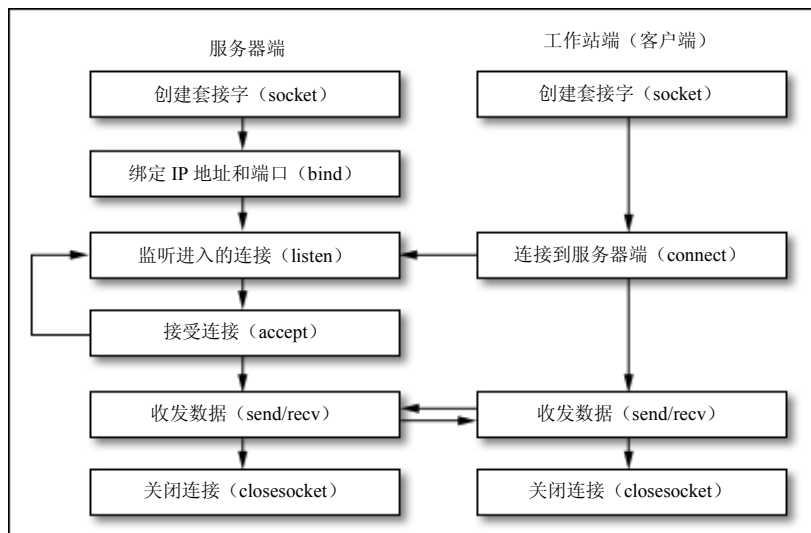


图 16.4 TCP 服务器端和客户端模型

就像打电话一样，A 对 B 说：“某某时候 Call 我”，那么 B 给 A 打电话的过程就可以用这种客户端/服务器端模型来描述。为了等待 B 的电话，A 必须在双方约定的某个特定的电话旁边等待，否则 B 将不知道如何 Call 他。由于“特定的电话”意味着服务器端的地址必须是特定的，所以服务器端的套接字必须首先使用 bind 函数绑定到指定的 IP 地址和端口上。因为“等待”意味着服务器必须随时监听客户端的连接动作，所以套接字绑定地址后必须使用 listen 函数进入监听状态。而对于 B 来说，他可以在任何时刻从任何电话给 A 打电话。由此可见，客户端使用的套接字并不需要绑定过程，让系统自动指定任意值并不影响它向服务器端发起连接。

客户端可以随时使用 `connect` 函数连接到服务器，服务器检测到这个连接后，需要使用 `accept` 函数接受这个连接，当服务器接受连接后，一个稳定的连接就建立了，双方就可以开始互相通过 `send` 和 `recv` 函数收发数据了，这时通信的两端并没有任何区别。

2. UDP 协议的特征和 UDP 程序的工作流程

虽然 TCP 协议由于可靠、稳定的特征而被用在大部分的应用场合，但是 UDP 协议由于对系统资源的要求比较小，所以也经常用在一些对数据的可靠性要求不高却要求效率的场所，如实时音频、视频点播和实时网络游戏等。

UDP 协议是一个无连接的，面向消息的，不可靠的传输层协议。

无连接的含义是：客户端在发送 UDP 数据包前不需要先与服务器端进行握手确认，不管服务器是否在线，是否已经准备接收 UDP 数据包，客户端都可以随时发送数据。由于 UDP 协议是无连接的，所以通过同一个 UDP 套接字可以向任何服务器地址发送数据，而不需要创建多个套接字。

面向消息的含义是：与 TCP 协议的数据流方式不同，UDP 数据包是有边界保护的，假设发送方分三次分别发送了 100、150、200 字节的 UDP 数据包，接收方必须分三次接收这些数据包，各数据包之间的数据不会粘连。

UDP 协议是不可靠的，它并不对数据的可靠性与有序性等进行控制，由于 UDP 协议不是面向连接的，所以无法确认对方是否在线，也无法确认对方的指定端口是否在监听中，它直接将数据包按照指定 IP 地址和端口发出去，如果对方不在线的话，数据就会丢失。

由于 UDP 协议也不提供应答和重传机制，所以程序并不知道数据是否到达。同样，UDP 协议也没有为数据包标识序号，所以数据到达对端的顺序有可能是不对的，相同的数据也有可能多次到达。它属于一种“发出就不管”的协议。

如果说 TCP 协议像是打电话，两个人在通话中实时地控制通话的流程并保证内容的正确性，那么 UDP 协议就像是寄信，发信人虽然知道收信人的地址，但是他并不确定信件会不会安全抵达，也不知道收信人究竟有没有因为外出而收不到信。另外，如果发信人发了好几封信，这些信可能并没有按发信时的顺序到达。总之，在收到收信人的回信之前，发信人无法确定信件是否安全、无损和有序地到达。

但是，正是因为 UDP 协议不对数据的可靠性进行保证，不需要在校验、排序、应答和流控上消耗资源，所以 UDP 协议的效率很高。在实际应用中，如果不需要很高的可靠性，使用 UDP 协议就非常适合，比如，对于实时视频，如果因为途中丢了数据包导致中断了几分之一秒，最重要的不是将它补上而是保证下面播放的实时性，如果为了补上丢失的数据包而导致“停格”显然是不必要的。

如图 16.5 所示，UDP 程序的客户端和服务端的工作方式区别不大，在客户端，UDP 套接字不必经过连接的过程就可以直接发送数据。在服务器端，UDP 套接字无所谓是否进入“监听”状态，只要有数据发送到套接字对应的端口，它就可以被接收，所以类似 TCP 程序架构中的连接（`connect`）、监听（`listen`）和接受连接（`accept`）等动作都是不存在的。服务器端和客

户端惟一的区别就是服务器端程序必须首先将套接字绑定到一个固定的端口，以便客户端能够向约定的端口发送数据。

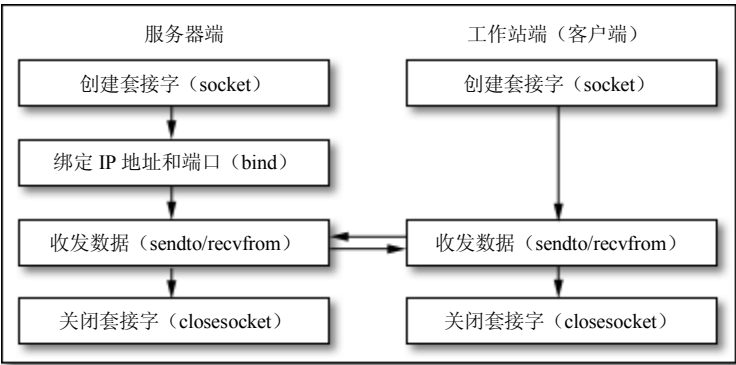


图 16.5 UDP 服务器/客户端模型

UDP 程序的流程相对简单，所以本章中不再详细举例对 UDP 的编程进行讲解。

16.2.4 监听、发起连接和接收连接

好了，到现在为止，读者已经了解了网络编程所需的各种基础知识，在下面的几节中，将详细介绍图 16.4 和 16.5 中提及的各种函数的详细用法。当然，本节的内容大部分是针对 TCP 套接字的，因为本节的主题是“监听、发起连接和接收连接”，而 UDP 套接字并不需要监听和连接操作，本节中仅有 bind 函数常用于操作 UDP 套接字。

1. TCP 客户端——连接到服务器

使用 TCP 套接字进行通信时，客户端必须首先使用 connect 函数连接到服务器的指定端口，connect 函数的用法是：

invoke	connect, s, lpsockaddr, len
--------	-----------------------------

参数 s 是 TCP 套接字的句柄，lpsockaddr 参数指向一个 sockaddr_in 结构，用来指定服务器端的地址和端口，len 参数则指定 sockaddr_in 结构的长度。

当套接字工作在阻塞模式下的时候，如果成功地连接到了服务器，那么函数返回 0，否则返回 SOCKET_ERROR。如果要知道连接失败的详细原因，可以马上调用 WSAGetLastError 来得到具体的出错代码。常见的错误有：当服务器端没有在指定端口监听时，出错代码是 WSAECONNREFUSED；如果网络不通，或者服务器不在线，那么错误代码可能是 WSAETIMEDOUT。

由于 TCP 连接的过程需要进行三次握手，也就是说应答的数据包需要在网络上传递三次，所以连接的过程视网络速度需要几十毫秒到几秒钟。当服务器端没有反应的时候，connect 函数可能会在十几秒钟后才因超时而失败返回，如果要在在这个过程中退出，可以在另外一个线程中使用 closesocket 关闭套接字。

当套接字工作在非阻塞模式下的时候，不管连接成功与否，connect 函数会马上返回并返回 SOCKET_ERROR，这时并不意味着连接失败，而是表示函数返回的时候连接尚未成功（请记住

非阻塞模式要求无论操作是否成功，函数必须马上返回）。只有调用 `WSAGetLastError` 函数得到的出错代码不是 `WSAEWOULDBLOCK`（表示出错原因是因为操作正在进行中）的话，才表示连接失败。

发起连接时，系统会自动为套接字选择一个空闲的端口，如果一定要用特定的端口连接到服务器端，可以在调用 `connect` 函数前先用后面介绍的 `bind` 函数将套接字绑定到指定的端口上。

2. TCP 服务器端——在指定的 IP 地址和端口监听并接收连接

不管是 TCP 套接字还是 UDP 套接字，如果希望套接字在指定的 IP 地址和端口监听，必须首先将它绑定到该 IP 地址和端口上，使用 `bind` 函数可以进行绑定操作：

invoke	<code>bind, s, lpsockaddr, len</code>
--------	---------------------------------------

参数 `s` 指定套接字句柄，`lpsockaddr` 参数指向定义了需要绑定的 IP 地址和端口的 `sockaddr_in` 结构，`len` 参数指定 `sockaddr_in` 结构的长度。

`sockaddr_in` 结构中的 `sin_port` 参数填写为需要进行监听的端口，`sin_addr` 参数只要填写成 `INADDR_ANY`（定义为 0）就可以了，表示自动在本机的所有 IP 地址上进行监听，例如，计算机有 3 个网卡，配置了 3 个 IP 地址，那么套接字会自动在所有 3 个地址上进行监听。

如果只需要在其中的某一个 IP 地址进行监听，不希望在全部地址监听怎么办？例如，服务器有一块网卡是连接因特网的，另一块网卡是连接内部网的，现在为了安全起见需要仅仅对内网的地址进行监听。当然可以，这时将 `sin_addr` 字段填写成内网的 IP 地址即可，套接字会在指定的那个地址上进行监听。

绑定成功的话，`bind` 函数返回 0，否则返回 `SOCKET_ERROR`。一般来说，绑定失败是因为端口已经被其他的程序占用了，这时的出错代码会是 `WSAEADDRINUSE`，还有一种情况是套接字已经绑定过了，那么错误代码会是 `WSAEFAULT`。

将套接字和指定的 IP 地址和端口绑定成功后，UDP 套接字就马上可以用来接收数据包了。但是 TCP 套接字还需要用 `listen` 函数进行监听，并使用 `accept` 函数接收进入的连接后才能进行通信。

使用 `listen` 函数可以使 TCP 套接字进入监听状态：

invoke	<code>listen, s, backlog</code>
--------	---------------------------------

参数 `s` 指定套接字句柄，`backlog` 参数是监听队列中允许保持的尚未处理的最大连接数量。注意不要将 `backlog` 参数理解为最多有多少个客户端能够和服务端相连接，它的真正含义是：当套接字监听到有客户端的连接请求后，还需要调用 `accept` 函数来接受连接，连接才真正被建立。在调用 `accept` 函数之前，连接请求将暂时被保存在队列中，如果这时有另一个客户端也发起连接的话，这个连接也将被保留在队列中，`backlog` 参数指的就是这个队列的最大长度。实际上，如果程序能够在足够短的时间内响应进入的连接，那么即使将 `backlog` 参数指定为 1，仍然不会丢失任何连接请求。

如果 `listen` 函数执行成功，将返回 0，这时套接字就处于等待连接进入的状态了。执行失

败，将返回 SOCKET_ERROR，有一种可能是还没有进行 bind 操作就去 listen，这时的出错代码是 WSAEINVAL。

当有客户端向监听中的 TCP 套接字发起连接后，必须对监听的套接字调用 accept 函数以后，连接才最后被确定：

```
invoke    accept, s, lpsockaddr, lpaddrlen
.if      eax != INVALID_SOCKET
    mov    hNewSocket, eax
.endif
```

参数 s 指定监听中的套接字句柄，lpsockaddr 指向一个缓冲区，函数会在这里返回一个 sockaddr_in 结构，结构中存放有连接请求方的 IP 地址和端口，lpaddrlen 则指向一个双字变量，函数在这里放入返回到缓冲区中的结构长度。如果不需要得到对方的地址信息，可以将 lpsockaddr 和 lpaddrlen 参数都设置为 NULL。

如果执行成功，函数将新建一个 TCP 套接字并返回它的句柄，这个新套接字才是和客户端相连接的，程序可以使用它来和客户端之间收发数据，原来在监听状态的套接字仍然保持监听状态，以便接受下一个连接的进入，如果函数执行失败，将返回 INVALID_SOCKET。

在这里读者一定要分清监听套接字和新套接字之间的区别。假如用于监听的套接字是 #1，那么前面的 bind, listen 和 accept 等函数都是对 #1 操作的，当 accept 函数返回套接字 #2 后，#2 才是和客户端相连的，所以为了和客户端进行通信而使用的 send 和 recv 等函数都是针对 #2 的。如果连接被客户端断开或者服务器主动断开连接，那么需要对 #2 调用 closesocket。只有服务器端程序想不再继续监听的时候，才需要对 #1 调用 closesocket 函数。

由于在没有客户端发起连接请求的时候，accept 函数不会返回，所以监听和接收连接的循环一般是这样写的：

```
invoke    listen, hListenSocket, 5
.while    TRUE
    invoke    accept, hListenSocket, NULL, 0
    .break    .if eax == INVALID_SOCKET
    ;在这里创建一个新线程对新的套接字进行通信，以便马上能处理新连接
    ...
.endw
```

在循环中，accept 函数成功返回时就意味这一个新的连接产生，但是在循环中直接对新连接进行数据收发是不恰当的，这样就没法马上回到 accept 函数处，其他客户端的连接请求就无法被处理了，所以一般创建一个新的线程来负责和新连接的通信工作，而循环马上返回到 accept 处等待新的连接。新的套接字句柄可以通过 lParam 参数传递给线程过程。

如果套接字不需要再进行监听了，可以在另一个线程中用 closesocket 函数将它关闭，这样，accept 函数会马上退出并返回 INVALID_SOCKET，程序就可以退出循环。

在使用 accept 函数的时候，还有一个简单的技巧：用第二个参数得到包含客户端 IP 地址和端口的 sockaddr_in 结构后，如果程序希望对客户端 IP 地址进行认证，那就可以在 accept 后马上进行判断，如果检测到 IP 地址不合法，则立刻使用 closesocket 函数将 accept 函数产

生的新套接字关闭，这样连接马上会被断开。

16.2.5 数据的收发

对于 TCP 连接来说，一旦连接已经建立（对客户端来说就是 connect 返回成功，对服务器端来说是 accept 函数返回了新连接的套接字句柄），那么连接的双方实际上是对等的，因为 TCP 连接是一个全双工的连接，任何一方都可以在任何时刻向对方发送数据。

1. 使用 TCP 套接字收发数据

在使用 TCP 连接向对方发送数据包时，一般使用 send 函数：

invoke send, s, lpbuf, len, flags

s 参数指定套接字句柄，lpbuf 指向包含要发送数据的缓冲区，len 参数指定发送的数据长度，flags 参数指定选项，这个参数一般指定为 0。

当函数返回时，如果发送失败，则返回值是 SOCKET_ERROR，否则返回值是函数发送成功的字节数（WinSock 接口为每个套接字分配一个发送缓冲区和一个接收缓冲区，用 send 函数发送数据时，数据并没有马上在网络上进行传递，而是首先放到 WinSock 接口模块的发送缓冲区中，接口会在合适的时候将数据发送出去。所以前面的“成功发送”指的是成功地放入发送缓冲区而已）。

函数在阻塞模式和非阻塞模式下的表现有些不同，下面以用 send 函数发送 n 字节的数据，而当前发送缓冲区的空闲空间是 m 字节的情况为例说明。

在阻塞模式下，当发送缓冲区足够空，也就是 $n \leq m$ 的情况下，这时函数会将数据全部放入发送缓冲区，然后马上返回；而缓冲区不够大，也就是 $n > m \geq 0$ 的时候，函数会一直等待，直到全部数据放入缓冲区为止。在两种情况下，返回值都是发送的实际字节数 n 。

而在非阻塞模式下，当发送缓冲区足够空，也就是 $n \leq m$ 的情况下，这时函数也会将数据全部放入发送缓冲区，然后马上返回，这时返回值就是发送的实际字节数 n ；当缓冲区不够大，也就是 $n > m > 0$ 的时候，函数不会等待，而是直接将 m 字节的数据放入缓冲区后马上返回，这时返回值是发送的实际字节数 m ；当发送缓冲区满，也就是 $m = 0$ 的时候，函数也会马上返回，这时返回值是 SOCKET_ERROR，并且如果用 WSAGetLastError 获取出错代码的话会得到 WSAEWOULDBLOCK。

所以在函数发送成功（也就是返回值不是 SOCKET_ERROR）时，阻塞模式下函数肯定是已经将全部 n 字节数据发送成功并返回 n ，但是期间可能会有个等待的过程。而在非阻塞模式下，函数肯定会马上返回，但函数实际发送的字节数可能会在 1 到 n 之间，不一定等于程序要求发送的字节数，所以程序以后必须对没有发送的部分进行重发。

而函数返回 SOCKET_ERROR 时，阻塞模式下肯定意味着连接已经因为各种情况而断开，而在非阻塞模式下，要继续用 WSAGetLastError 获取出错代码，如果得到的出错代码是 WSAEWOULDBLOCK 意味着缓冲区满（这时实际发送 0 字节），否则表示连接已经断开了。

在使用 TCP 连接接收数据包时，一般使用 recv 函数：

 invoke recv, s, lpbuf, len, flags

s 参数指定读取的套接字句柄, lpbuf 指向一个用来返回数据的缓冲区, len 参数指定缓冲区的大小, flags 参数用来指定读取时的选项, 它可以是 MSG_PEEK 和 MSG_OOB 的组合, MSG_PEEK 表示返回数据后并不从缓冲区中清除数据。

当函数返回时, 如果接收失败, 则返回值是 SOCKET_ERROR, 否则返回值是函数实际接收的字节数 (WinSock 接口收到数据时, 首先会将数据放入接收缓冲区中, 用 recv 函数接收数据实际上是从缓冲区中取数据)。

同样, 函数在阻塞模式和非阻塞模式下的表现有些不同, 下面以用 recv 函数接收 n 字节的数据, 而当前接收缓冲区中有 m 字节数据的情况为例说明。

在阻塞模式下, 如果接收缓冲区为空, 即 $m=0$ 的情况下, 则函数等待直到有数据到达为止; 如果缓冲区中已经有 m 字节的数据 (或者缓冲区为空时, 函数等待, 然后有 m 字节的数据到达后), 如果 $m \geq n$, 则函数从缓冲区中取 n 字节的数据并返回; 如果 $m < n$, 那么函数只取 m 字节的数据并马上返回。函数的返回值是实际接收到的字节数, 这个值会在 1 到 n 之间。也就是说在不超过 n 字节的前提下, 有多少数据到达函数即返回多少数据。

在非阻塞模式下, 当接收缓冲区中已经有数据的情况下, recv 的表现方式和阻塞模式相同, 即函数会马上返回, 并视缓冲区中数据的数量返回 1 到 n 字节之间的数据; 但是在缓冲区为空, 即 $m=0$ 的情况下, 函数不会等待, 而是马上返回 SOCKET_ERROR, 这时得到的出错代码会是 WSAEWOULDBLOCK。

所以在函数成功返回 (也就是返回值不是 SOCKET_ERROR) 时, 两种模式下函数都不一定得到请求接收的字节数, 实际接收的字节数可能会在 1 到 n 之间。如果要收满一定数量的数据才能进行下一步操作的话, 程序必须循环接收直到接收的总数据量达到指定值为止。

而函数返回 SOCKET_ERROR 时, 阻塞模式下肯定意味着连接已经因为各种情况而断开, 而在非阻塞模式下, 要继续用 WSAGetLastError 获取出错代码, 如果得到的出错代码是 WSAEWOULDBLOCK 意味着缓冲区空 (这时实际接收 0 字节), 则表示连接已经断开了。

读者必须仔细体会两种模式下 send 和 recv 函数的不同表现, 因为在具体的应用中, 函数的不同表现足以影响程序的架构设计。

2. 使用 UDP 套接字收发数据

UDP 协议不是面向连接的, 当 UDP 套接字被创建以后, 就可以直接通过它向服务器端发送数据了, 由于前面介绍的 send 函数没有目标地址参数, 所以一般不用它来发送 UDP 数据报, 取而代之的是 sendto 函数:

 invoke sendto, s, lpbuf, len, flags, lpto, tolen

参数 s 指定用来发送数据的套接字句柄, 参数 lpbuf 指向包含发送数据的缓冲区, 参数 len 指定要发送的数据的长度, 参数 flags 一般指定为 0, 参数 lpto 指向一个包含目标地址和端口号的 sockaddr_in 结构, 参数 tolen 指定了这个结构的大小。

TCP 套接字在连接成功后，就只能向连接的对方发送数据，不可能途中再换一个 IP 地址和端口重新进行连接，而 UDP 套接字不是面向连接的，使用同一个 UDP 套接字，每次调用 `sendto` 函数的时候指定不同的服务器端 IP 地址和端口，就可以向不同的服务器发送数据。

UDP 数据包有最大尺寸 `SO_MAX_MSG_SIZE` 的限制，如果发送的数据包小于该尺寸，并且发送成功，那么函数将返回实际发送数据的字节数；如果数据包大小超过该尺寸，函数将返回失败，这时没有任何数据被发送，并且出错代码是 `WSAEMSGSIZE`。

`sendto` 函数在阻塞模式和非阻塞模式下也有所不同，阻塞模式下如果没有足够的发送缓冲区，函数将等待到缓冲区空出足够的大小为止；而非阻塞模式下，函数会马上返回 `SOCKET_ERROR`，这时得到的出错代码会是 `WSAEWOULDBLOCK`。由于 UDP 协议是面向消息的，数据包不会被割裂发送，所以不管哪种情况下，函数不会只发送部分数据。

用 `sendto` 发送 UDP 数据包时，发送方使用哪个端口呢？实际上，对一个 UDP 套接字首次调用 `sendto` 函数时，系统会自动分配一个空闲的端口，在这后面再调用 `sendto` 函数，系统将一直使用分配的这个端口。所以对于 UDP 套接字来说，如果在对它调用 `sendto` 函数之前没有用 `bind` 函数绑定到一个固定的 IP 地址或端口上，意味着这时它还没有被系统自动指派某个端口，就无法使用它来接收数据（还没有地址和端口如何接收数据呢？），直到第一次对它使用 `sendto` 函数来发送数据以后，系统才自动将它绑定到某个空闲的端口上，这时套接字才可用来接收数据。

接收 UDP 数据包一般使用 `recvfrom` 函数：

invoke	<code>recvfrom, s, lpbuf, len, flags, lpfrom, lpfromlen</code>
--------	--

参数 `s` 指定套接字句柄，参数 `lpbuf` 指向一个用来接收数据的缓冲区，参数 `len` 指定缓冲区的大小。参数 `flags` 为标志。这个函数不同于 `recv` 函数的是多出来的最后两个参数，参数 `lpfrom` 指向一个缓冲区，函数在这里返回一个包含数据发送方地址的 `sockaddr_in` 结构，参数 `lpfromlen` 指向一个双字变量，函数在这里返回前面的 `sockaddr_in` 结构的长度。

同样，由于 UDP 套接字不是面向连接的，所以使用同一个 UDP 套接字可以接收从任何客户端发送过来的 UDP 数据包，只要对方指定了正确的 IP 地址和端口。

程序可以从 `sockaddr_in` 结构中得到发送方的 IP 地址和端口，如果需要向发送方回复数据的话，可以根据这个地址和端口来回复。

UDP 协议是面向消息的，当使用 `recvfrom` 函数接收时，如果指定的缓冲区尺寸小于接收缓冲区中 UDP 包的尺寸，那么缓冲区中将收到部分数据，多余的部分会丢失，这时函数将返回 `SOCKET_ERROR`，具体的出错代码是 `WSAEMSGSIZE`。如果缓冲区大于要接收的 UDP 包的尺寸，系统仅仅返回要接收的 UDP 数据包，而不会将后面到达的数据包的内容一并返回。这时函数的返回值是实际接收的数据大小。

函数在阻塞模式和非阻塞模式下也有所不同，阻塞模式下接收缓冲区如果没有数据到达，函数将等待到有数据包到达为止；而非阻塞模式下，函数会马上返回 `SOCKET_ERROR`，这时得到的出错代码会是 `WSAEWOULDBLOCK`。

哭

575

576

器器

```

                pop     ecx
                invoke  CloseHandle, eax
;*****
                .elseif eax == WM_CLOSE
                invoke  closesocket, hListenSocket
                or      dwFlag, F_STOP
                .while  dwThreadCounter
                .endw
                invoke  WSACleanup
                invoke  EndDialog, hWinMain, NULL
;*****
                .else
                mov     eax, FALSE
                ret
                .endif
                mov     eax, TRUE
                ret

_ProcDlgMain    endp
;>>>>>>>>>
; 程序开始
;>>>>>>>>>
start:
                invoke  GetModuleHandle, NULL
                invoke  DialogBoxParam, eax, DLG_MAIN, \
                        NULL, offset _ProcDlgMain, 0
                invoke  ExitProcess, NULL
;>>>>>>>>>
                end     start

```

程序的工作流程大致如下：

首先，程序创建了一个对话框作为主界面，在对话框的 WM_INITDIALOG 消息中，调用 WSAStartup 加载 WinSock 库，然后创建一个线程来运行负责监听和接收连接的循环。

在调用 CreateThread 函数的时候，程序使用了一个小技巧，由于 CreateThread 函数的最后一个参数是 lpThreadId，得到的 ThreadId 没有继续使用的价值，所以没有必要专门为它定义一个变量。程序用 push ecx 指令临时在堆栈中分配一个 dword 空间供其使用，调用时使用 esp 将该临时空间的地址传给函数即可，然后在函数返回的时候直接用一个 pop ecx 操作释放堆栈空间（这个用法和子程序在堆栈中为局部变量保留空间的用法很像，不是吗？）。

监听线程的线程过程是 ListenThread，程序在这里首先用 socket 函数创建一个 TCP 套接字，并将套接字绑定在 9999 端口上，后面的流程就是 listen 和调用 accept 的循环了。

每次有客户端发起连接时，accept 函数返回一个新的套接字，这个套接字就是与客户端连接的套接字，程序将创建一个新的线程 ServiceThread 来负责与客户端进行通信。所以，每次有新的连接进入，服务器端程序的进程中就会多一个线程，而关闭一个连接后，就会少一个线程，这一点读者可以从任务管理器中得到验证。accept 产生的新套接字句柄通过线程过程的参数传递给 ServiceThread 线程。

在 ServiceThread 的开始和最后，程序在对话框中显示当前的在线客户端数量，在子程序

的中间，是一个 select 函数→recv 函数→send 函数的循环，让我们先忽略掉 select 函数，来看看 recv 和 send 函数吧。

套接字创建的时候，默认是工作在阻塞模式下，程序中没有将它设置成非阻塞模式，所以请读者努力回忆一下，recv 和 send 函数在阻塞模式下是怎么工作的。

想起来了么？在阻塞模式下，recv 函数在接收缓冲区中没有数据的时候会等待，直到有数据到达为止，虽然程序在参数中指定接收 sizeof @szBuffer 个字节，但是函数不一定等到收满 sizeof @szBuffer 字节后才返回，接收到的数据数量和客户端发送过来的数据量有关（会从 1 到 sizeof @szBuffer 之间），具体要以返回值为准。

然后就是用 send 函数把收到的数据发送回去，以完成 Echo 的功能，注意要发送的字节数就是前面 recv 函数的返回值，由于阻塞模式下 send 函数返回时肯定是将请求发送的数量全部发送完毕，所以在这个例子中程序不需要做进一步处理。但是读者应该知道，要是在非阻塞模式下完成同样的 send 操作，程序必须判断返回值，当返回值小于请求发送的字节数时，需要循环将剩余的字节全部发送出去。

循环中对 recv 和 send 函数的返回值进行了出错判断，一旦函数返回 SOCKET_ERROR，表示连接已经中断，程序将退出循环，然后关闭套接字，线程终止。

好了，现在我们来看看 select 函数是做什么用的，读者请首先设想一下，程序中要是把 select 函数去掉，仅仅剩下 recv 和 send 函数的循环会怎样？那样的话，如果客户端没有发送数据的时候，程序就会等待在 recv 函数中，这时按下服务器端对话框中的关闭按钮的话，线程将无法检测到退出标志（dwFlag 设置为 F_STOP）并退出。如果非要退出的话，在对话框的 WM_CLOSE 消息中就必须关闭每个线程中的套接字句柄，这样 recv 函数才会出错退出，但这样的话程序要维护一个在线连接的列表，会复杂很多。

反过来，如果先用某种方法检测是否有数据到达，有的话才去 recv，那么就不会有上述情况发生了，select 函数就是这样用的，这个函数可以用来检测套接字的各种情况，以便程序根据检测结果做下一步操作。

select 函数可以同时检测多个套接字，可以检测套接字是否可读、是否可写，以及是否有异常发生：

```
invoke    select, nfds, lpreadfds, lpwritefds, lpexceptfds, lptimeout
```

参数 nfds 是为了和 BSD UNIX Socket 的兼容而设置的，WinSock 接口下函数将这个参数忽略，lpreadfds, lpwritefds 和 lpexceptfds 分别指向不同的 fd_set 结构，用来指定需要检测的套接字句柄。fd_set 结构的定义如下：

```
fd_set STRUCT
    fd_count  DWORD    ?                ; fd_array 中存放的套接字句柄数量
    fd_array  DWORD FD_SETSIZE dup(?)    ; 套接字句柄列表
fd_set ENDS
```

假如要检测 #1 和 #2 套接字是否可读（也就是接收缓冲区中已经有数据到达），那么可以在 lpreadfds 参数指向的 fd_set 结构的 fd_array 中放入它们的句柄并将 fd_count 设置为 2。

如果同时要检测 #1、#3 和 #4 套接字是否可写（也就是发送缓冲区是否有空），那么可以在 `lpwritefds` 指向的 `fd_set` 结构中的 `fd_array` 中放入它们的句柄并将 `fd_count` 设置为 3。同样，`lpeexceptfds` 指向的 `fd_set` 结构存放的是要检测出错（如检测连接是否断掉）的套接字句柄列表。如果没有某种操作需要检测，那么可以将相应的指针设置为 `NULL`。

当函数返回的时候，函数会将这些 `fd_set` 结构中就绪的套接字句柄保留，而将没有就绪的套接字句柄清零，这时扫描结构中的 `fd_array` 列表并对还存在的套接字句柄进行相应的操作即可。

`select` 函数的 `lptimeout` 参数指向一个 `timeval` 结构，用来指定检测的超时时间，该结构定义如下：

```
timeval STRUCT
    tv_sec DWORD    ?      ;秒
    tv_usec DWORD    ?      ;微秒，注意不是毫秒！
timeval ENDS
```

`lptimeout` 参数的用法有以下几种：

- 如果 `lptimeout` 参数为 `NULL`，那么函数将永远等待下去，直到列表中有某个套接字就绪时才返回。
- 如果 `lptimeout` 指向了一个 `timeval` 结构，而且结构中的时间定义为 0，那么不管有没有套接字就绪函数都会马上返回。
- 如果 `lptimeout` 指向了一个 `timeval` 结构，而且结构中的时间定义不为 0，如果经过了 `timeval` 指定的时间后还没有套接字就绪，那么函数超时返回；如果在指定的时间有套接字就绪，那么函数马上返回。

当 `select` 函数因为超时而返回时，返回值是 0；因为出错而返回时，返回值是 `SOCKET_ERROR`；如果因为某个套接字就绪而返回，返回值是就绪套接字的数量。

在例子中，程序每隔 200ms 对套接字进行检测，由于要检测的套接字数量为 1，所以返回值大于 0 的时候，肯定就是这个套接字有数据可以接收了，程序马上进行 `recv` 和 `send` 操作。如果返回值是 0，表示过了 200ms 还没有数据可以接收，这时程序跳过 `recv` 和 `send` 操作，回到循环的开始去检测循环条件，当检测到退出标志被设置时，循环结束，套接字被关闭，然后线程终止。

当按下了对话框中的关闭按钮时，程序在 `WM_CLOSE` 消息中将监听的套接字句柄关闭，这样，监听线程中的 `accept` 函数会失败返回，监听线程终止；接下来程序设置退出标志，如果有客户端在线的话，对应的工作线程会检测到退出标志并终止，等所有工作线程终止后（检测线程计数为 0），程序用 `WSACleanup` 释放 WinSock 库并退出。

16.3 TCP 应用程序的设计

虽然 `TcpEcho` 例子的目的只是让读者实践一下各函数的使用方法，但这个例子示范的程序

结构却是 TCP 服务器端最常用的架构，其特征是用一个独立的线程进行监听和接收连接，然后为每个客户端创建一个工作线程。这种架构的好处是工作流程很清晰，与客户端相关的数据（如登录信息等，这类数据在别的系统中一般称为 Session 或者会话）放在线程过程的局部变量中，不必集中维护，这样就回避了在全局变量中维护 Session 列表时涉及的多线程数据同步问题。

当然，这种架构的缺点就是客户端数量到达一定程度时，由于创建的工作线程过多，浪费在线程切换上的时间比较多，造成性能上的下降。基本上，同时在线的客户端数量应该控制在几百个以下，如果同时在线的客户端数量远大于几百个，最合适的方法是使用完成端口（I/O Completion Port, IOCP）模型。当然，在客户端数量没有达到这个规模时，IOCP 模型的复杂性带来的缺点反而会彻底抵消掉带来的好处。对 IOCP 有兴趣的读者可以自行查看相关的资料。

除了网络游戏或者大型网站的服务器端，由于大部分的网络服务程序同时在线客户端数量都不会超过百个，所以 TcpEcho 采用的程序结构是非常流行的。读者只需沿用程序的架构并将工作线程进行功能上的扩展，就可以实现复杂的通信服务。

本节讨论的内容就是如何设计通信的流程，以便完成复杂而完善的网络服务功能，另外还要讨论的是设计中的注意事项，很少有资料提及这些细节，但是要设计无差错并且“强壮”的商业化网络服务程序，这些细节都是必须要考虑到的。

本节中会用一个 TCP 聊天室的例子来演示相关的内容，在最后的小节中还会将聊天室客户端改成以非阻塞模式工作，以便讲解非阻塞模式下的程序结构。

16.3.1 通信协议和工作线程的设计

1. 设计 TCP 链路的通信协议

很少有网络服务程序完成的功能像 TcpEcho 例子这么简单，大部分的 TCP 程序是采用应答方式的，客户端会向服务器端发送不同的命令请求，服务器端根据请求做不同的操作，并把结果返回给客户端。例如，一个远程控制程序的服务器端会提供列目录、拷贝文件、运行文件、删除文件、关闭计算机等各种功能，客户端发送的数据中会包含命令代码，服务器端收到以后根据代码做对应的操作，并将结果返回。

这种工作模式下，客户端需要将命令代码和相关数据组成一个个数据包发送到服务器端，但是各命令的数据包长度是不同的，比如，运行文件时还要提供运行的文件名，而关机操作就只需要一个命令码，既然客户端是以不同长度的命令数据包为单位进行发送的，那么服务器端如何正确地以命令数据包为单位进行接收呢？

读者可能会说，那还不简单，根据阻塞模式下 recv 函数的特征，将缓冲区设置为足够大，并在 recv 函数的接收字节数参数中指定足够大的值，这时函数并不会将缓冲区收满才返回，而是客户端发过来多少数据，就会收到多少数据，既然客户端一次发一个命令数据包，服务器端不就可以每次收到一个完整的数据包吗？

可是现实却不是如此，前面介绍过，TCP 协议提供的是流服务，其含义就是协议不对数据包进行边界保护，发送端的 WinSock 接口会根据缓冲区和数据包的实际情况，自由地对数据包进行组合和分割发送，也就是说，客户端连续发送多个数据包的时候，多个数据包可能会“粘

作者曾经分析了网上下载到的很多个实现聊天功能的例子代码，发现绝大多数例子使用的都是这种错误的接收方法，这种方法在网络非常通畅、通信双方的负载很低的情况下是正确的，所以用于演示目的不会发现什么异常，一旦投入到网络情况复杂、服务器负荷很重的情况下使用，就会出现串包现象。很明显，要书写任何情况下都是无错的程序，不考虑数据包的粘连和分割现象是不行的。

```
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 命令代码定义
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
CMD_LOGIN          equ      01h        ; 客户端 ->服务器端，登录
CMD_LOGIN_RESP     equ      81h        ; 服务器端 -> 客户端，登录回应
CMD_MSG_UP         equ      02h        ; 客户端 -> 服务器端，聊天语句
CMD_MSG_DOWN       equ      82h        ; 服务器端 -> 客户端，聊天语句
CMD_CHECK_LINK     equ      83h        ; 服务器端 -> 客户端，链路检测
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据包定义
; *****
; 数据包头部，所有的数据包都以 MSG_HEAD 开头
; *****
MSG_HEAD            struct
    dwCmdId           dw        ?             ;命令 ID
    dwLength          dd        ?             ;整个数据包长度=数据包头部+数据包体
MSG_HEAD            ends
; *****
; 登录数据包（客户端->服务器端）
; *****
MSG_LOGIN           struct
    szUserName        db        12 dup (?)   ;用户名 ID
    szPassword        db        12 dup (?)   ;登录密码
MSG_LOGIN           ends
; *****
; 登录回应数据包（服务器端->客户端）
; *****
MSG_LOGIN_RESP      struct
    dbResult          db        ?             ;登录结果：0=成功，1=用户名或密码错
MSG_LOGIN_RESP      ends
; *****
; 聊天语句（客户端->服务器端）：不等长数据包
; *****
MSG_UP              struct
    dwLength          dd        ?             ;后面内容字段的长度
    szContent         db        256 dup (?)  ;内容，不等长，长度由 dwLength 指定
MSG_UP              ends
```

[illegible]

文件的开头部分定义了命令代码，这一点并没有什么特殊之处，聊天室例子程序支持的命令代码有：客户端登录、服务器端对登录的回应、客户端发送聊天语句，以及服务器端转发其他客户端的聊天语句等。

后面就是对数据包的定义了，正确的设计方法是和数据包的组成分成两部分——数据包头和数据包体，读者接下来马上就可以发现这种设计的奥妙。

数据包头一般包含两个字段：命令代码字段和数据包长度字段，例子中的 MSG_HEAD 结构就是数据包头，里面有 dwCmdId 和 dwLength 字段。数据包体的定义则根据命令的不同而不同，比如，登录数据包体 MSG_LOGIN 结构中包含登录的用户名和密码，登录回应数据包 MSG_LOGIN_RESP 结构中只有表示登录是否成功的状态字段。文件的最后用 MSG_STRUCT 结构定义完整的数据包，这个结构表现了完整数据包的组成方式——结构的第一项是一个 MSG_HEAD 结构，第二项是一个包含各种数据包体定义的集合。

这样定义有什么好处呢？好处就是便于接收方从数据流中解出正确的数据包。数据包头的长度是确定的，接收方每次接收的时候首先收全一个数据包头部（假如一次调用 `recv` 得到的数据少于数据包头的长度，则循环 `recv` 直到收全一个数据包头），然后从数据包头中的 `dwLength` 字段中得到整个数据包的确切长度，将这个长度减去包头的长度后，剩下的就是需要接收的数据包体长度，然后就可以循环调用 `recv` 函数，等收全整个数据包后就可以处理了。

TCP 聊天室例子目录下还有一个_SocketRoute.asm 文件, 其中的_RecvPacket 子程序就是用上面描述的方法接收一个完整的数据包的, 请读者看看这个文件的内容:

```
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 在规定的时间内等待数据到达
; 输入： dwTime = 需要等待的时间（微秒）
; >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_WaitData proc    _hSocket, _dwTime
                local   @stFdSet:fd_set, @stimeval:timeval
```

584


```

_RcvPacket    proc        _hSocket, _lpBuffer, _dwSize
               local      @dwReturn

               pushad
               mov         @dwReturn, TRUE
               mov         esi, _lpBuffer
               assume      esi:ptr MSG_STRUCT
;*****
; 接收数据包头部并检测数据是否正常
;*****
               invoke      _RecvData, _hSocket, esi, sizeof MSG_HEAD
               or          eax, eax
               jnz         _Ret
               mov         ecx, [esi].MsgHead.dwLength
               cmp         ecx, sizeof MSG_HEAD
               jb          _Ret
               cmp         ecx, _dwSize
               ja          _Ret
;*****
; 接收余下的数据
;*****
               sub         ecx, sizeof MSG_HEAD
               add         esi, sizeof MSG_HEAD
               .if         ecx
                   invoke   _RecvData, _hSocket, esi, ecx
               .else
                   xor      eax, eax
               .endif
               mov         @dwReturn, eax
_Ret:
               popad
               assume      esi:nothing
               mov         eax, @dwReturn
               ret

_RcvPacket    endp

```

在_RcvPacket 子程序中，程序首先调用_RcvData 子程序接收一个数据包头，然后对数据包头进行适当的校验，假如里面的 dwLength 字段内容小于一个包头的长度，或者大于缓冲区的长度，那么说明数据包是不合法的，程序将出错返回；如果通过了校验，那么继续调用_RcvData 子程序接收数据包的剩余部分，这部分长度等于 dwLength 减去 MSG_HEAD 的长度。

_RecvData 子程序循环调用 recv 函数来收全指定字节数的数据，由于发送方因为各种原因只发送部分数据时（比如，连上来的是不合法的程序，并没有按照规定的协议发送整个数据包），会造成程序长时间阻塞在 recv 函数中，所以在调用 recv 函数前，要先使用 select 函数检测是否有数据到达。子程序中每次还用 GetTickCount 函数获取时间值并检测接收数据已经花费的总时间，如果超过了规定的时间则作为错误处理。

由于这两个子程序实现的功能很常用，所以例子中将它们单独放到_SocketRoute.asm 源文件中，以便同时在服务器端和客户端中使用。读者在编写其他网络应用程序的时候，也可以将其

包含在源代码中直接调用，这时只需要修改涉及数据包头定义的几句代码即可。

当然，在实际的应用中设计的通信协议不一定就是和上面的例子一模一样的，这里有几个变通设计的例子。

假如服务器端和客户端都已经明确约定各个命令的数据包长度，那么包头的定义里面就可以没有 `dwLength` 字段，因为接收方根据包头里面的命令码也可以查出包体的长度是多少，但是，明确地用一个 `dwLength` 字段指定整个数据包的长度有很多好处。

第一个好处是有利于兼容性，假设服务器端的协议升级导致某个命令数据包的长度改变了，这样老版本的客户端连上来时发过来的数据包长度就会不符，服务器端凭命令码判断而收取的数据包长度是不对的，后面的数据包就会错位，而总是根据 `dwLength` 字段的长度收数据包则在任何时刻都不会发生数据包错位的现象；第二个好处是定义 `dwLength` 字段有利于简单的纠错，因为这个字段的值小于包头的长度或者大到超过缓冲区长度都表示数据包有误；还有一个好处是便于定义不定长的数据包，以节省网络带宽，例如，数据包里包含一个文件名时，只需发送文件名字符串的有效字符就可以了，没必要非得发完固定的 `MAX_PATH` 长度。

数据包头里面也可以增加其他一些字段，如命令的流水号、数据包的校验和、数据包头的特征标识等。例如，定义数据包头的第一个字段是个单字节的特征标识，这个字段的内容必须是 `0FFh` 才是合法的，那么接收方收到第一个字节不是 `0FFh` 的数据包，就可以得知数据包有误。

不管怎样对通信协议的设计进行变通，请读者记住一个要点，那就是数据包的结构中必须有一个固定长度的包头，便于接收方首先接收，其次包头中必须包含整个数据包的长度信息（或者从包头里面的其他数据可以推断出整个数据包的长度），这样接收方才能从数据流中正确分解出完整的数据包。



当然，如果读者要编写的不是基于命令应答的应用，就无须以本节描述的方法定义通信协议。比如，基于数据流的应用就根本不存在分解数据包的问题，典型的基于数据流的应用是从网上下载一个文件，接收方只要连续接收并存盘即可。

2. 链路异常检测

在 TCP 应用程序的设计中，一个很重要但又经常被忽略的问题是对链路异常情况的检测和处理，什么是链路异常呢？读者可以先用前面的 `TcpEcho` 例子程序做一个实验：

准备两台连网的计算机（注意，不要通过直连的对绞线相连，两台计算机中间要至少经过一个 HUB），在计算机 A 上运行 `TcpEcho` 程序，然后在计算机 B 上用 `telnet xx.xx.xx.xx 9999` 连接到 A 计算机，这时 `TcpEcho` 的对话框上可以看到在线客户端数量为 1。现在模拟网络中断或者一方的计算机崩溃的情况，将计算机 B 的网线拔掉（如果读者不心疼的话，也可以直接拔计算机 B 的电源线！），可以发现，`TcpEcho` 对话框上显示的在线客户端数量还是 1。

理论上，连接已经中断了，那么等多久 `TcpEcho` 程序才能检测到这一情况呢？很不幸，如果读者有足够的耐心等待的话，就可以发现等上几天几夜，在线客户端数量还是 1。也就是说，这个无效的工作线程一直在占用资源。由于网络故障和客户端死机的情况是不可能完全避免

的，如果服务器端程序是个长年在线服务的程序，一段时间后，这种“僵尸”线程就会越来越多，直到消耗完所有的资源，造成程序崩溃为止。

程序无法检测网络断线的原因与 TCP 协议的工作原理有关，TCP 连接正常断开的时候，主动断开的一方会向另一方发送含有断开连接标志的数据包，接收方收到数据包后，将连接的状态更新为断开，任何对套接字的操作就会失败，如果这时有 `recv` 函数或者 `select` 函数在对套接字进行阻塞等待，函数就会马上返回，返回值是 `SOCKET_ERROR`。所以连接正常断开的时候，工作线程是可以检测到的。

但是出现网络故障或者一方的计算机崩溃时，另一方是收不到含有断开连接标志的数据包的，系统只会认为对方一直没有数据过来，这种情况要延续到程序需要主动发送数据包为止，如果主动发送的数据包得不到对方的应答，在经过几次尝试全部超时以后，系统就会意识到连接已经断开。

更确切地说，连接异常中断后的首次 `send` 调用是会返回成功的，因为这时仅仅表示数据成功地放入了发送缓冲区，WinSock 接口还没开始在网络上发送数据包，所以还没法检测到连接已中断，在经过几分钟后，由这个 `send` 调用引发的数据包发送动作一直得不到对方的回应，WinSock 才将连接的状态置为断开，以后对套接字的操作才会全部失败。

所以发现这种链路异常的惟一办法是主动发送数据，但很多服务器端程序仅仅处理客户端的请求后才回送数据，从来不会主动向客户端发送数据，上面的 `TcpEcho` 例子就是如此，所以程序根本没有机会检测到链路异常。

要解决这种问题，必须在通信协议的设计中就考虑到这一点，常用的方法是通信双方都记录链路的最后一次活动时间（比如，收到过数据或者发送过数据），一旦空闲的时间到一定的秒数，就由一方主动向另一方发一个数据包。由于这个数据包仅仅是为了检测链路而发，一般将其称为“链路检测包”，链路检测包没必要定义包体，只需要一个包头即可。例如，前面的例子中定义了一个链路检测命令编码 `CMD_CHECK_LINK` equ 83h，但并没有定义一个链路检测包体的结构。

一旦链路异常，发送方在发送链路检测包后不久就可以检测到；对于接收方，如果在规定的空闲时间内收到链路检测包，那么只需更新链路的最后一次活动时间，并不需要做额外的操作，如果在规定的时间内没有收到，就可以认为链路异常中断了。一般来说，接收方的时间要定义得稍微宽余一些，比如，规定发送方在链路空闲 30 秒后发送链路检测包，那么接收方可以在链路空闲 60 秒后才认为连接异常，否则因为定时的误差，数据包已经在路上了，接收方却认为时间已到而退出，就起不到应有的作用了。在具体的使用中，由服务器端还是由客户端发送链路检测包并没有规定，只需从编程方便的角度考虑就可以了。

3. 多线程下的数据收发

在实际编程中，经常要遇到在不同的线程中操作同一个套接字的问题，比如，设计阻塞模式工作的客户端的时候，需要循环调用 `select` 函数来检测是否由数据到达，由于在主线中实现循环很不方便，所以往往单独创建一个线程，在线程中循环调用 `select` 函数和 `recv` 函数来接收数据。另一方面，由于发送数据的动作都是由用户在界面上操作引发的，所以 `send` 函

数在主线程中调用更方便。读者在下面的 TCP 聊天室客户端例子中就可以看到，接收操作是在单独的接收线程中完成的，而发送操作是在主线程中通过响应“发送”按钮的动作实现的。

那么跨线程对同一个套接字进行收发操作究竟有没有什么限制呢？这要从 WinSock 接口对 `recv` 和 `send` 函数的处理方式来判断。

WinSock 接口对发送缓冲区和接收缓冲区的操作有排队锁定机制，也就是说，当多个线程同时调用 `send` 函数对同一个套接字进行发送操作时，只有一个线程能锁定发送缓冲区，其余线程处于等待状态，当活动线程的 `send` 函数调用完毕后，再轮到其他线程的 `send` 函数。所以同时在多个线程中调用 `send` 函数，这些函数发送的数据在网络上也不会交织在一起。

同样，多个线程同时调用 `recv` 函数对同一个套接字进行接收操作时，也只有一个线程能锁定接收缓冲区，其余线程处于等待状态，也就是说，即使多个线程同时调用 `recv` 函数，同一份数据也不会被多个线程重复接收。

仔细体会 WinSock 接口的处理方式，结合前面介绍的数据包头加数据包体的封装方式，再结合处理数据包的 `_RecvPacket` 子程序的工作流程，就可以分析出在多线程中对同一个套接字进行收发的可能性。

首先分析一下发送数据包的情况。

在阻塞模式下，`send` 函数总是将指定大小的数据包发送完才返回，由于多个 `send` 函数发送的数据不会交织，所以只要每次调用 `send` 函数发送的数据包都是一个完整的数据包的话，就不必担心数据包之间的数据会互相交织。但是对一个数据包分开发送的话（比如，先调用一次 `send` 函数发送数据包头，再调用一次 `send` 函数发送数据包体）就可能出错，因为线程 A 发送的数据包头发送完毕后，可能接下去发送的是线程 B 的数据包头，然后才轮到线程 A 的数据包体，这样接收方收到的数据就是错误的。

而在非阻塞模式下，即使一次指定发送一个完整的数据包，函数也可能只发送部分数据，程序需要判断还有多少数据没有发送，并循环调用 `send` 函数将整个数据包发完。这样，循环调用 `send` 的过程中可能会被其他线程的 `send` 操作插入，造成不同数据包的数据交织在一起。所以非阻塞模式下不能有多个线程同时对一个套接字进行发送操作。如果一定要在多个线程中同时发送，那么需要采用临界区等对象进行线程同步，以保证发送一个完整的数据包的期间不会被其他线程打断。

接收数据包时，不管是阻塞模式还是非阻塞模式下，`recv` 函数总是不能保证一次收全一个数据包，而且 `_RecvPacket` 子程序的实现中，也必然是分多次调用 `recv` 函数完成的（至少两次，一次接收数据包头，一次接收数据包体），这就意味着一个线程分多次收取一个数据包的时候，中间的部分数据可能被其他线程收走，造成数据错误。所以两种模式下，都不能有多个线程同时去收数据包，如果一定要在多个线程中接收数据包，也需要采用临界区等对象进行线程同步，以保证接收一个完整的数据包的期间不会被其他线程打断。

如图 16.6 所示，一般来说，TCP 连接上传递的数据包的应答情况有三种，最简单的情况如图中左边部分所示，服务器端和客户端根据自己的需要在任意时刻发送数据包，对方收到数据包以后不需要回复。如图中客户端分别发送了 C1、C2 和 C3 共三个数据包，而服务器端发送了

S1 和 S2 两个数据包，这些数据包之间并没有关联。这样，收发操作既可以在一个线程中实现，也可以在多个线程中实现。比如，主线程负责发送数据包，另一个线程只负责接收数据包。

图 16.6 中中间部分所示的是单方向的应答式通信方式——客户端发送命令（如 C1、C2 数据包），服务器收到数据包后进行处理，并返回结果（C1_RESP 和 C2_RESP），客户端发完一个命令数据包后，必须等到对应的回应信息后，才能继续发送后一个命令。而服务器端不主动向客户端发送数据包。在这种情况下，由于客户端中发送命令的代码后面紧跟着接收的代码（服务器端的顺序相反），一般将收发操作放在同一个线程中实现。

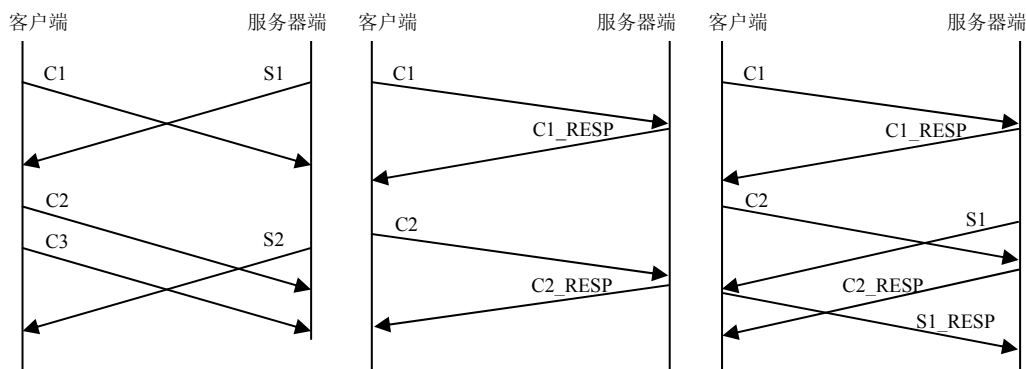


图 16.6 网络应用程序的常见通信方式

图 16.6 中右边部分所示的是双方互为应答的通信方式——在单方向应答式通信的基础上，服务器端也可能主动向客户端发送命令请求，客户端处理后返回结果数据包，这样双方的程序在逻辑上有两部分，一个部分是等待对方的命令，进行处理并返回；另一个部分是发送命令，等待对方的处理并返回。这两个部分必须放在同一个线程中来实现。原因就是，两个部分都有接收数据包的操作，而我们知道，在多个线程中同时接收数据包可能会产生错误。而且，即使分两个线程，然后对 _RecvPacket 子程序用临界区对象进行同步也是不行的，如图 16.6 的右边所示，假设在线程 A（主动发送命令的线程）中发送了 C2 数据包，这时程序预期等待的是 C2_RESP 数据包，但实际接收的可能是本该由线程 B（等待命令并处理的线程）接收并处理的 S1 数据包，虽然可以保证收到的是完整的数据包，但是程序的处理流程就全部乱套了。

在实际的使用中，往往在同一个线程中用下面的代码结构处理互为应答式的通信：

```

...
;*****
; 处理接收的命令并返回处理结果的模块，输入参数为接收的命令数据包
;*****
_ProcRecvCmd      proc
    .if            命令码 == C1
                    处理并用 send 发送 C1_RESP 数据包
    .elseif        命令码 == C2
                    处理并用 send 发送 C1_RESP 数据包
    .elseif        ...

```

```

        .endif
        ret
    _ProcRecvCmd    endp
;*****
; 主动发送命令并接收处理结果的模块
;*****
_SendCmd  proc

        .if          命令队列中没有命令需要发送
            ret
        .endif
        从队列中取需要发送的命令
        .if          命令码 == S1
            用 send 发送 S1 数据包
            @@:
            用_RcvPacket 接收回复的数据包
            .if      回复数据包 != S1_RESP
                invoke    _ProcRecvCmd
                jmp       @B
            .endif
        .elseif      命令码 == S2
            用 send 发送 S2 数据包
            @@:
            用_RcvPacket 接收回复的数据包
            .if      回复数据包 != S2_RESP
                invoke    _ProcRecvCmd
                jmp       @B
            .endif
        .elseif      ...
        .endif
        ret

_SendCmd  endp
;*****
; 工作线程
;*****
        ...
        .while      TRUE
            invoke    _SendCmd
            invoke    _CheckLine    ;发送链路检测包（具体程序省略）
            调用 select 函数等待 100ms，查看是否有数据到达
            .if      有数据到达
                调用_RcvPacket 接收整个数据包
                invoke    _ProcRecvCmd
            .endif
        .endw

```

在这种架构下，需要建立一个发送命令队列，所有线程产生的发送命令请求统一放到发送队列中，然后在工作线程中调用_SendCmd 子程序（发送模块）去检测发送命令队列，如果队列不为空，则进行发送操作并等待对方的回应数据包。_SendCmd 子程序的关键是，如果在发送了某个命令后，接收到的不是预期的回应数据包，就表示这个数据包是对方主动发送的命令，这时程序先调用_ProcRecvCmd 子程序对这个命令进行处理并回复，然后继续等待预期的回应数据

包，直到等待规定的回应数据包后才退出。

在循环中，发送模块处理完毕后，程序用 `select` 函数检测是否有数据包到达，如果有的话，则接收一个数据包，然后调用 `_ProcRecvCmd` 子程序进行处理。由于前面的 `_SendCmd` 子程序中对每个发送的命令都会先接收对应的回应数据包后再退出，所以可以保证这里收到的数据包肯定对方发送的命令数据包。命令处理完毕后，再进行下一轮循环。

当然，循环中间还有很多的异常处理代码，例子中只列出了对链路检测包的示范，实际应用中，在每次的发送和接收语句后面，都应该检测函数的返回值，如果出错的话则退出循环并关闭套接字，然后终止线程。

16.3.2 TCP 聊天室例子——服务器端

好了，读者现在应该对 TCP 应用程序的设计思路和注意事项有了相当的了解，说得多不如做得多，现在我们用一个 TCP 聊天室的例子来实践一下基于数据包应答的网络应用程序。详细的源代码位于光盘的 `Chapter16\Chat-TCP` 目录下，其中服务器端程序用到了以下文件：

- `Server.rc`——服务器端程序的资源脚本文件。
- `Server.asm`——服务器端程序的汇编源代码。
- `_Message.inc`——通信协议的定义文件，同时供服务器端和客户端包含使用。
- `_SocketRoute.asm`——阻塞模式下通用的子程序，在源代码中包含使用。
- `_MsgQueue.asm`——实现消息队列的几个子程序，在源代码中包含使用。

在这些文件中，`_Message.inc` 文件和 `_SocketRoute.asm` 文件的作用在上一个节中就已经详细分析过了，`_MsgQueue.asm` 文件则是用于维护一个消息队列的，为什么程序中还要用到消息队列呢？

这是因为：在大部分的聊天室例子中，当服务器端收到某个客户端的聊天语句时，会用一个循环将这条语句转发给所有在线的其他客户端，这样，如果要转发的客户端中有些网速很慢，所有客户端的聊天速度都会被拖慢，因为程序必须等转发的循环结束才能接收下一条聊天语句；另外，为了将语句转发给所有在线的其他客户端，程序必须维护一个在线客户端列表，但是随着客户端的频繁登录退出，对列表进行增删操作是件很麻烦的事情。

为了解决这些矛盾，程序设计了一个消息队列，当服务器端收到某个客户端的聊天语句后，不向其他在线的客户端主动转发，而是将聊天语句插入消息队列，消息队列是先进先出（FIFO）类型的，当队列满的时候，最早的一条消息被挤出队列，队列中的消息按顺序被定义了从小到大的消息编号。

某个工作线程将消息放入队列后，其他客户端的工作线程在接收聊天语句的空闲时间中，主动获取队列中的消息并进行发送。这样，程序就不必维护一个在线客户端列表供循环发送使用。各个工作线程从消息队列中获取了一条消息并发送后，记录下已经发送的消息编号，下一次就从这个编号的后一条消息开始获取，如果队列中没有更高编号的消息，意味着所有的聊天语句都已经转发过了；如果要获取的编号比队列中最小的消息编号都要小，意味着网速太慢导

通过这种方式，系统的工作会以正常的节奏进行，速度过慢的客户端不会影响其他客户端的工作，而是会丢失一些聊天语句，但这样更符合正常的使用习惯。

其中的 `_InsertMsgQueue` 子程序在队列中加入一条消息，如果队列已经满了，则将整个队列前移一个位置，相当于最早的消息被覆盖，然后在队列尾部空出的位置加入新消息；如果队列未满，则在队列的最后加入新消息。`_GetMsgFromQueue` 子程序则从队列获取指定编号的消息，如果指定编号的消息已经被清除出消息队列，则自动返回编号最小的一条消息；如果队列中的所有消息的编号都比指定编号小（意味着所有消息都被获取过了），那么不返回任何消息。

服务器端工作线程的具体实现方式如下:

[illegible]

器器

594

```

;*****
        .while    !(dwFlag & F_STOP)
            invoke    _SendMsgQueue, _hSocket, esi, edi
            .break    .if eax
            invoke    _LinkCheck, _hSocket, esi, edi
            .break    .if eax
            .break    .if dwFlag & F_STOP
;*****
; 使用 select 函数等待 200ms, 如果没有接收到数据包则循环
;*****
            invoke    _WaitData, _hSocket, 200 * 1000
            .break    .if eax == SOCKET_ERROR
            .if      eax
                invoke    _RecvPacket, _hSocket, esi, sizeof @szBuffer
                .break    .if eax
                invoke    GetTickCount
                mov     [edi].dwLastTime, eax
                .if     [esi].MsgHead.dwCmdId == CMD_MSG_UP
                    invoke    _InsertMsgQueue, \
                                addr [edi].szUserName, \
                                addr [esi].MsgUp.szContent
                .endif
            .endif
        .endw
;*****
; 广播: xxx 退出了聊天室
;*****
            invoke    lstrcpy, esi, addr [edi].szUserName
            invoke    lstrcat, esi, addr szUserLogout
            invoke    _InsertMsgQueue, addr szSysInfo, addr @szBuffer
;*****
; 关闭 socket
;*****
_Ret:
            invoke    closesocket, _hSocket
            dec     dwThreadCounter
            invoke    SetDlgItemInt, hWinMain, IDC_COUNT, \
                                dwThreadCounter, FALSE
            popad
            ret

_ServiceThread    endp
                assume    esi:nothing, edi:nothing
...

```

首先, 程序定义了一个 SESSION 结构, 用来保存每个会话 (Session) 的数据。所谓会话, 就是服务器端为每个连接上来的客户端分别保存的各种信息, 比如, 工作状态、套接字句柄、登录用户名等各连接之间互相独立的数据。在本例中, SESSION 结构中定义了三个字段: szUserName 为客户端登录时使用的用户名; dwMessageId 表示已经向客户端转发的最后一条消息编号; dwLastTime 表示链路空闲的时间, 用于定时发送链路检测包。

每产生一个客户端连接的时候都需要分配一个结构用于保存会话数据, 直到客户端断开的时

候才被释放，如果集中管理会话数据的话，不管是采用预留 n 个 SESSION 结构的内存空间，还是动态申请空间的方法，都会涉及空间的申请、查找和释放的问题，实现起来比较麻烦。

对于为每个客户端连接产生一个工作线程的架构来说，将 SESSION 结构放在工作线程的局部变量中分散管理是最方便的，这样在线程开始的时候，SESSION 结构自动被分配，线程结束的时候会被自动释放，本例中采用的就是这种方式。

在工作线程中，完成对 SESSION 结构的初始化工作后，马上要处理的是客户端的登录操作，客户端会首先发送一个 Login 数据包，包头中的命令代码是 CMD_LOGIN，服务器端必须检测收到的第一个数据包是否是登录数据包，如果不是的话即关闭连接。

假如成功地收到了 Login 数据包，服务器端将给客户端回复一个包含 CMD_LOGIN_RESP 命令的 LoginResp 数据包。在本例中程序做了简化，任何的密码都能通过登录，而且程序也不对同样的用户名进行重复登录的检测，在实际的应用中，程序可以在这个阶段查询保存在数据库或者文件中的用户名和密码进行对比，如果登录信息错误，则在回复的 LoginResp 包中填写错误信息通知客户端，并主动断开连接。登录成功后，程序将 Login 数据包中的用户名保存到 SESSION 结构中，以便以后使用。

由于在客户端进入收发聊天语句的循环前必须登录，例子中将登录数据包的应答工作放在主循环前完成，这样程序的结构更加合理和容易理解。登录验证通过后，程序向消息队列中插入“某某已登录”的消息，然后进入聊天语句的收发循环。

循环采用的架构正是 16.3.1 节末尾介绍的，回过头来复习一下这个架构：

```

while TRUE
    invoke    主动发送数据包的模块
    invoke    链路超时检测模块
    调用 select 函数等待 100ms，查看是否有数据到达
    .if      有数据到达
        调用 _RecvPacket 接收整个数据包
        invoke 命令处理模块
    .endif
.endw

```

在例子中，主动发送数据包的模块对应的是 _SendMsgQueue 子程序，子程序中循环从消息队列中获取消息并发送到客户端，直到队列中不再有新的消息为止。考虑到发送速度比较慢的时候，队列中会一直有消息等待发送而导致程序一直在 _SendMsgQueue 子程序中打转，没有机会退到外层循环中接收并处理数据包，所以每次发送数据包后用 select 函数（在 _WaitData 子程序中）检测一下，假如有数据到达则优先退出并处理到达的数据包。

链路超时检测模块则由 _LinkCheck 子程序实现，子程序中对 SESSION 中保存的 dwLastTime 变量进行检测，一旦计数值和当前时间的差值超过 30 秒，则尝试发送一个命令编号为 CMD_CHECK_LINK 的链路检测包。

由于例子中的命令处理模块比较简单，所以没有写成一个单独的子程序，而是直接写在了循环中，处理方法就是将收到的聊天数据包用 _InsertMsgQueue 子程序添加到消息队列中，以供其他工作线程获取并转发。

读者还可以注意到，不管是在主动发送数据包的模块还是接收数据包的模块中，凡是成功发送或者接收数据后，程序总是用 `GetTickCount` 函数获取当前时间计数并更新 `SESSION` 结构中的 `dwLastTime` 字段，这样连接足够忙碌的时候，`_LinkCheck` 子程序中的时间差就永远不会超过 30 秒，也就不必发送链路检测包。只有连接过于空闲的时候，才会每隔 30 秒发送一次链路检测包。

程序中还在每个发送和接收函数后进行错误检测，一旦客户端主动断开或者连接异常中断，循环就能马上退出，退出后程序在消息队列中插入一条“某某退出聊天室”的语句后，关闭套接字并终止线程。

需要一提的是，程序对不定长数据包的处理，从 `_Message.inc` 文件中可以看到，上行和下行的聊天语句数据包 `MSG_UP` 和 `MSG_DOWN` 结构中都有 `szContent` 字段，这个字段是按照最长 256 字节定义的，但实际聊天的时候很少有这么长的语句，如果每次发送数据包的时候都按照最长长度发送，对网络带宽的浪费是很严重的，所以程序中对这两个数据包的发送都采用了不定长的方式。

invoke	lstrlen, addr [esi].MsgDown.szContent
inc	eax
mov	[esi].MsgDown.dwLength, eax
add	eax, sizeof MSG_HEAD+MSG_DOWN.szContent
mov	[esi].MsgHead.dwLength, eax
mov	[esi].MsgHead.dwCmdId, CMD_MSG_DOWN
invoke	send, _hSocket, esi, [esi].MsgHead.dwLength, 0

具体的算法如上面的语句所示，程序首先使用 `lstrlen` 函数计算字符串的长度，加上 1 就是算上字符串结束符 0 的长度。在 MASM 的语法中，“结构名. 字段名”代表该字段在结构中的偏移量，所以 `sizeof MSG_HEAD+MSG_DOWN.szContent` 的值就是数据包头的长度加上结构中 `szContent` 之前的所有字段的长度，这个值加上 `szContent` 字段中字符串的总长度，就是整个数据包的长度。

如果读者有兴趣尝试一下对程序进行扩展的话，可以思考一下下面的问题（当然，真正开始行动必须等看完客户端的代码后再开始，因为网络程序的客户端和服务端总是需要互相配合工作的）。

- （1）如何在下行的聊天消息中加上用户发言的时间？
- （2）如何限制用户灌水，即短时间内发布大量的重复语句？
- （3）如何加上对用户重复登录的限制？
- （4）如何实现悄悄话的功能？

对于这些问题的解决方法，这里有些简单的建议，希望能够以此扩展读者的思路。

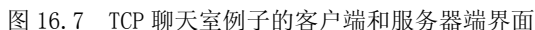
第一个问题的实现是最简单的，首先在消息队列和下行的 `MSG_DOWN` 结构中分别加上发言时间就可以了，其次工作线程收到聊天语句后，需要获取当前时间，并将其和其他数据一起放入消息队列。

解决第二个问题要复杂一点，需要对 `SESSION` 结构进行扩展，增加记录上次发言时间的字

第四个问题的解决应该建立在解决第三个问题的基础上，首先定义 SEND_ONLINE_USER 数据包，在客户端登录后，服务器端即主动向客户端发送该数据包，数据包中包含在线用户列表数据，客户端收到后即可用一个下拉框显示在界面上；其次是客户端需要跟踪“某某用户登录”，以及“某某用户退出”的消息，以便更新在线用户下拉框；再次，MSG_UP 消息中需要增加发送对象字段，用于指定向全部用户发送还是向某个用户发送，服务器端将该信息一并放入消息队列；最后，每个工作线程取消息的时候要判断消息中的发送对象，如果是发往其他用户的“悄悄话”，则不进行转发。

现在来看看和服务端配套的客户端例子。

客户端运行的界面如图 16.7 所示。



客户端对应的资源脚本文件 Client.rc 内容如下:

[illegible]

以阻塞模式工作的客户端汇编源代码 Client1.asm 的内容如下:

599

600


```

        invoke    htons, TCP_PORT
        mov       @stSin.sin_port, ax

        invoke    socket, AF_INET, SOCK_STREAM, 0
        mov       hSocket, eax
;*****
; 连接到服务器
;*****
        invoke    connect, hSocket, addr @stSin, sizeof @stSin
        .if       eax == SOCKET_ERROR
            invoke    MessageBox, hWinMain, addr szErrConnect, \
                    NULL, MB_OK or MB_ICONSTOP
            jmp      _Ret
        .endif
;*****
; 登录到服务器
;*****
        invoke    lstrcpy, addr @stMsg.Login.szUserName, \
                addr szUserName
        invoke    lstrcpy, addr @stMsg.Login.szPassword, \
                addr szPassword
        mov       @stMsg.MsgHead.dwLength, \
                sizeof MSG_HEAD+sizeof MSG_LOGIN
        mov       @stMsg.MsgHead.dwCmdId, CMD_LOGIN
        invoke    send, hSocket, \
                addr @stMsg, @stMsg.MsgHead.dwLength, 0
        cmp       eax, SOCKET_ERROR
        jz        @F
        invoke    _RecvPacket, hSocket, addr @stMsg, sizeof @stMsg
        or        eax, eax
        jnz       @F
        cmp       @stMsg.MsgHead.dwCmdId, CMD_LOGIN_RESP
        jnz       @F
        .if       @stMsg.LoginResp.dbResult
            @@:
            invoke    MessageBox, hWinMain, addr szErrLogin, \
                    NULL, MB_OK or MB_ICONSTOP
            jmp      _Ret
        .endif

        invoke    GetDlgItem, hWinMain, IDC_LOGOUT
        invoke    EnableWindow, eax, TRUE
        invoke    GetDlgItem, hWinMain, IDC_TEXT
        invoke    EnableWindow, eax, TRUE
        invoke    GetTickCount
        mov       dwLastTime, eax
;*****
; 循环接收消息
;*****
        .while    hSocket
            invoke    GetTickCount
            sub       eax, dwLastTime
            .break    .if eax >= 60 * 1000 ; 链路空闲时间检测
            invoke    _WaitData, hSocket, 200 * 1000
        .endwhile

```

602

```

        .if      (ax == IDC_SERVER) || (ax == IDC_USER) || (ax == IDC_PASS)
        invoke   GetDlgItemText,hWinMain, IDC_SERVER, \
                addr szServer, sizeof szServer
        invoke   GetDlgItemText,hWinMain, IDC_USER, \
                addr szUserName, sizeof szUserName
        invoke   GetDlgItemText,hWinMain, IDC_PASS, \
                addr szPassword, sizeof szPassword
        invoke   GetDlgItem, hWinMain, IDC_LOGIN
        .if      szServer && szUserName && szPassword && !hSocket
        invoke   EnableWindow, eax, TRUE
        .else
        invoke   EnableWindow, eax, FALSE
        .endif
;*****
; 登录成功后，输入聊天语句后才激活“发送”按钮
;*****
        .elseif ax == IDC_TEXT
        invoke   GetDlgItemText,hWinMain, IDC_TEXT, \
                addr szText, sizeof szText
        invoke   GetDlgItem, hWinMain, IDOK
        .if      szText && hSocket
        invoke   EnableWindow, eax, TRUE
        .else
        invoke   EnableWindow, eax, FALSE
        .endif
;*****
        .elseif ax == IDC_LOGIN
        push     ecx
        invoke   CreateThread, NULL, 0, offset _WorkThread, 0, NULL, esp
        pop      ecx
        invoke   CloseHandle, eax
;*****
        .elseif ax == IDC_LOGOUT
        @@:
        .if      hSocket
        invoke   closesocket, hSocket
        xor      eax, eax
        mov      hSocket, eax
        .endif
;*****
; 单击“发送”按钮则构造 MSG_UP 数据包并发送到服务器
;*****
        .elseif ax == IDOK
        invoke   lstrcpy, addr @stMsg.MsgUp.szContent, addr szText
        invoke   strlen, addr @stMsg.MsgUp.szContent
        inc      eax
        mov      @stMsg.MsgUp.dwLength, eax
        add      eax, sizeof MSG_HEAD+MSG_UP.szContent
        mov      @stMsg.MsgHead.dwLength, eax
        mov      @stMsg.MsgHead.dwCmdId, CMD_MSG_UP
        invoke   send, hSocket, addr @stMsg, @stMsg.MsgHead.dwLength, 0
        cmp      eax, SOCKET_ERROR
        jz       @B
        invoke   GetTickCount
        mov      dwLastTime, eax

```

```

        invoke SetDlgItemText,hWinMain,IDC_TEXT,NULL
        invoke GetDlgItem,hWinMain,IDC_TEXT
        invoke SetFocus,eax
    .endif
;*****
    .elseif eax == WM_CLOSE
        .if ! hSocket
            invoke WSACleanup
            invoke EndDialog,hWinMain,NULL
        .endif
;*****
    .elseif eax == WM_INITDIALOG
        push hWnd
        pop hWinMain
        invoke WSASStartup,101h,addr @stWsa
        invoke SendDlgItemMessage,hWinMain,\
            IDC_SERVER,EM_SETLIMITTEXT,15,0
        invoke SendDlgItemMessage,hWinMain,\
            IDC_USER,EM_SETLIMITTEXT,11,0
        invoke SendDlgItemMessage,hWinMain,\
            IDC_PASS,EM_SETLIMITTEXT,11,0
        invoke SendDlgItemMessage,hWinMain,\
            IDC_TEXT,EM_SETLIMITTEXT,250,0
;*****
    .else
        mov     eax,FALSE
        ret
    .endif
    mov     eax,TRUE
    ret

_ProcDlgMain    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 程序开始
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
start:
        invoke GetModuleHandle,NULL
        invoke DialogBoxParam,eax,DLG_MAIN,\
            NULL,offset _ProcDlgMain,0
        invoke ExitProcess,NULL
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    end        start

```

从通信的角度来看，客户端并不复杂，接收操作是在一个单独的线程中完成的，发送操作是在主线程中响应“发送”按钮的代码中完成的。程序的大部分代码用于在通信流程中对各按钮和控件的状态进行修改。

在运行后，初始状态下用户可以输入服务器 IP 地址、用户名和密码，这时其他的所有按钮都是灰化的，只有全部输入这三个参数后，“登录”按钮才被激活。三个文本框的检测代码在对 WM_COMMAND 消息的响应中实现。

输入 IP 地址、用户名和密码后，再按下“登录”按钮，程序将创建一个单独的线程来连接到服务器并与服务器进行通信。线程过程是 _WorkThread，请读者注意，其中对按钮状态的

更新代码，线程中首先将 IP 地址、用户名、密码输入框和“登录”按钮全部灰化，以防止“登录”按钮被多次按下导致创建多个工作线程。

接下来在线程中用 `socket` 函数创建套接字、填写 `sockaddr_in` 结构并用 `connect` 函数连接到服务器，一旦连接成功，程序首先向服务器端发送一个 Login 数据包，数据包中包含了用户输入的用户名和密码信息，然后用 `_RecvPacket` 子程序接收返回的 LoginResp 数据包，假如数据包中的 `dbResult` 字段为 0，表示登录成功（这个字段的含义是我们自己在通信协议中定义的，不是吗？），这时程序将“注销”按钮和聊天语句输入框激活，以便用户可以输入聊天语句或者从服务器注销。

在连接和登录的过程中如果出错，包括 IP 地址无法解析、无法连接到服务器或者服务器返回的 LoginResp 数据包中 `dbResult` 字段的代码不正确，程序将显示对应的出错提示消息框并终止线程，线程过程在退出前，IP 地址、用户名、密码输入框将被重新激活，以便用户重新进行连接和登录的操作。

成功登录到服务器后，工作线程即进入接收、处理数据包的循环，如果接收到 `MsgDown` 数据包，则将数据包中的发送者和内容字段组合成一个字符串显示在列表框中。

在 16.3.1 节中曾经分析过，由于阻塞模式下在不同线程中发送数据包不会有问题，所以用户输入聊天语句并按下“发送”按钮的时候，程序直接在窗口消息的处理代码中进行发送操作，如果发送失败，则表示连接中断，程序转到“注销”按钮的对应代码中去处理。

发送失败或者用户按下“注销”按钮的时候，按钮的处理代码中将套接字关闭，这样工作线程中的 `select` 函数或者 `recv` 函数的执行就会失败，工作线程将退出循环并终止线程。

工作线程中还加入了对链路异常中断的检测，程序定义了一个 `dwLastTime` 变量，每次成功接收或者发送数据包后，都会将 `dwLastTime` 变量更新为当前时间计数。每次循环中，都会将当前时间和 `dwLastTime` 变量的差值进行检查，在连接正常并且空闲的时候，服务器端每隔 30 秒会发过来一个链路检测包，所以正常情况下差值不会超过 30 秒，一旦检测到差值大于 60 秒，那么表示连接异常中断了，循环即退出。

16.3.4 以非阻塞方式工作的 TCP 聊天室客户端

到现在为止，前面的例子都是关于阻塞模式的，本节将介绍如何在 Windows 下以非阻塞方式对套接字进行收发操作，并将 TCP 聊天室例子中的客户端改为以非阻塞模式工作。

1. WinSock 接口非阻塞模式的工作方式

在前面的内容中已经详细介绍过各个函数在非阻塞模式下的表现方式，非阻塞模式的一个显著特点就是，任何 WinSock 函数在被调用后肯定是马上返回的，不会等待操作完成才返回。如果函数是因为阻塞而出错返回，那么错误代码是 `WSAEWOULDBLOCK`。

在 UNIX 下，阻塞方式和非阻塞方式的区分仅限于此，但如果仅仅是这点区别的话，只要将函数的出错代码做一下判断并稍微调整一下各函数的使用方式即可，没必要专门以一节的篇幅来说明。实际上，WinSock 接口对非阻塞方式的工作模式做了很大的扩展，最主要的特征是，

函数因阻塞而马上返回后，接口会在后台继续操作，一旦操作完成，会以窗口消息的方式通知窗口过程。因为这种特征，Windows 系统非阻塞模式下的网络应用程序结构将与阻塞模式的完全不同，阻塞模式下的程序架构是过程驱动的（即按顺序执行），而非阻塞模式下的程序架构是消息驱动的。

当一个套接字被创建的时候，它默认工作在阻塞模式下，用 `WSAAsyncSelect` 函数可以将套接字设置为非阻塞模式，并且打开套接字的消息通知机制，通知消息可以被绑定到某个窗口句柄中，这样程序就不必不停地去查询套接字的操作是否已经完成，只要在窗口过程中等收到通知消息后再进行处理即可。WSA 带头的函数是 WinSock 接口扩展的函数，在 UNIX 系统下并不存在。

`WSAAsyncSelect` 函数的用法是：

invoke	<code>WSAAsyncSelect, s, hWnd, wParam, lParam</code>
--------	--

`s` 参数指定需要设置的套接字句柄，`hWnd` 指定一个窗口句柄，套接字的通知消息将被发送到与其对应的窗口过程中。

通知消息的消息编号可以由程序自己定义，当不同的动作完成以后，不同的通知码将被包含在消息的参数中传递给指定的窗口过程，`wParam` 参数用来定义通知消息的编号，读者可以在 `WM_USER` 以上数值中任取一个。

最后的参数 `lParam` 指定哪些通知码需要发送，它可以被指定为几个通知码的组合，常用的通知码如下：

- `FD_READ`——套接字收到对端发送过来的数据包，表明这时可以去读套接字。
- `FD_WRITE`——当短时间内向一个套接字发送太多数据造成缓冲区满以后，发送函数会返回出错信息，当缓冲区再次有空的时候，WinSock 接口通过这个通知码通知应用程序，表示可以继续发送数据了。但是缓冲区未溢出的情况下，数据被发送完毕的时候并不会发送这个通知码。
- `FD_ACCEPT`——监听中的套接字检测到有连接进入（适用于 TCP 套接字）。
- `FD_CONNECT`——如果用一个套接字去连接对方主机，当连接动作完成以后将收到这个通知码（适用于 TCP 套接字）。
- `FD_CLOSE`——检测到套接字对应的连接被关闭（适用于 TCP 套接字）。

在使用中并不需要指定全部这些通知码。例如，UDP 套接字没有连接和断开的过程，所以 `FD_CONNECT`，`FD_CLOSE` 和 `FD_ACCEPT` 等通知码是没有意义的，而 TCP 套接字不用于监听的话，`FD_ACCEPT` 也是没有意义的，所以要根据套接字的类型选用合适的通知码。

窗口过程收到通知消息后，消息的 `wParam` 参数是触发消息的套接字句柄（可能有多个套接字绑定到同一个窗口中，这时可以用 `wParam` 参数加以区分），`lParam` 参数的低 16 位就是通知码，如果函数执行成功，`lParam` 的高 16 位为 0，执行失败的话，高 16 位是出错代码（相当于阻塞模式下调用了 `WSAGetLastError` 后得到的出错代码）。

WinSock 程序在非阻塞模式下的架构如图 16.8 所示，一般来说 WSAStartup 函数被安排在窗口的初始化消息中（图中的①），当程序需要退出时，程序调用 WSACleanup 函数卸载 WinSock 库（图中的⑧）。

当程序向服务器端发起连接时（比如，用户按下了“登录”按钮，图中的②），程序使用 socket 函数创建套接字，然后使用 WSAAsyncSelect 函数将通知消息以自定义的编号（图中以 WM_SOCKET 举例）绑定到窗口过程中，接下来用 connect 函数去连接服务器。

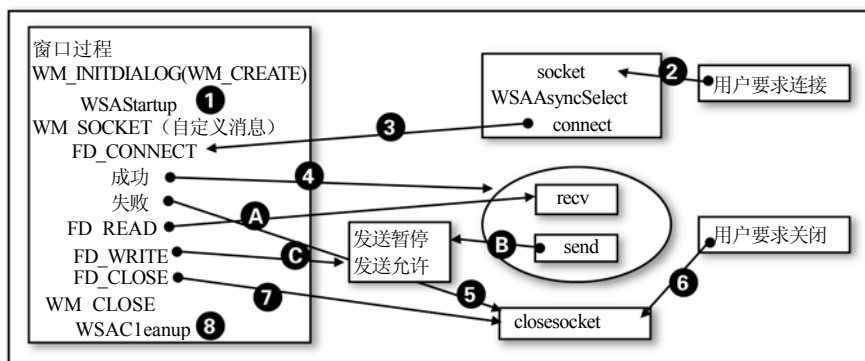


图 16.8 非阻塞模式下网络程序的常见结构

一般来说，非阻塞模式下的连接代码如下：

```
invoke    connect,hSocket,addr @stSin,sizeof @stSin
.if      eax == SOCKET_ERROR
    invoke WSAGetLastError
    .if   eax != WSAEWOULDBLOCK
        ;真正的出错，关闭套接字并退出
    .endif
.endif
```

connect 函数会马上返回，那么什么时候才知道连接的动作已经完成了呢？那就要等待包含 FD_CONNECT 通知码的 WM_SOCKET 消息了（图中的③），当收到通知消息时，lParam 参数的高 16 位包含了出错信息，如果检测到高 16 位等于 0 表示连接成功，可以开始收发数据了（图中的④）；否则表示没有连接成功，程序应该关闭套接字并显示出错信息（图中的⑤）。

FD_CONNECT 通知码的处理代码举例如下：

```
mov      eax,wMsg
.if     eax == WM_SOCKET
    mov   eax,lParam
    .if   ax == FD_CONNECT
        shr    eax,16    ;取 lParam 的高 16 位
        .if    !ax
            ;连接成功，可以进行收发数据了
        .else
            ;连接失败，显示出错信息
            ;并且关闭套接字
        .endif
    .elseif eax == ...
```

当从 `FD_CONNECT` 通知码的参数判断连接成功以后，就可以开始收发数据了，但是什么时候有数据可供接收呢，是不是需要随时去读套接字？答案是并不需要，如果有数据传过来，WinSock 接口会发送包含 `FD_READ` 通知码的窗口消息，程序可以在消息的处理代码中读取数据（图中的 A 所示）。

有一点要注意的是，如果收到 `FD_READ` 通知码后程序没有去读取数据，这时系统又收到了新的数据的话，那么系统并不会再次发送通知，而是必须在调用了 `recv` 函数以后才有可能再次收到 `FD_READ` 通知。但是，如果调用 `recv` 后没有读完接收缓冲区中的所有数据，系统会再发送一个 `FD_READ` 通知，表示缓冲区中仍有数据存在。

程序任何时候都可以调用 `send` 函数来发送数据，但是一旦得到 `WSAEWOULDBLOCK` 错误后，程序就应该暂停发送，因为这时表示发送缓冲区已满，即使不停尝试重发的话也会得到同样的错误。

那么什么时候可以再发送呢？答案是等到 `FD_WRITE` 通知后就可以了，程序可以设置一个标志和 `FD_WRITE` 通知配合起来控制流量，标志一开始被设置为“可以发送”状态，如果使用 `send` 函数发送数据时发现返回 `WSAEWOULDBLOCK` 错误，程序暂停发送并将标志设置为“暂停发送”（图中的 B 所示），等收到 `FD_WRITE` 通知后（图中的 C），程序可以将标志恢复为“可以发送”状态并继续发送数据。

断开连接的方式有两种，一种是在本地使用 `closesocket` 函数关闭套接字（图中的⑥），另一种是当连接被对方关闭时，系统会通过 `FD_CLOSE` 通知码来通知应用程序（图中的⑦），这时程序也应该将套接字关闭，因为这时套接字已经不能用来收发数据了。

2. 非阻塞模式工作的聊天室客户端

当 TCP 链路上收发的数据是数据流方式的话，非阻塞模式的程序架构还是非常方便的，但是链路上传输的是经过封装的数据包的时候，用上面的架构实现起来就比较麻烦了。下面以具体的例子来说明如何编写非阻塞模式下的 TCP 聊天室客户端。

例子程序的界面和 Client1 例子相同，使用的资源文件也是 Client.rc 文件，完整的汇编源代码见 Chapter16\Chat-TCP\Client2.asm 文件，其中和对话框界面有关的代码和 Client1.asm 例子是一样的，这里仅列出和通信相关的部分代码：

[illegible]

610

器器

612

613

```

        invoke EnableWindow, eax, TRUE
        invoke _SendData, 0, 0      ;继续发送缓冲区数据
    .elseif ax == FD_CONNECT
        shr    eax, 16
        .if    ax == NULL          ;连接成功则登录
            invoke lstrcpy, addr @stMsg.Login.szUserName, \
                addr szUserName
            invoke lstrcpy, addr @stMsg.Login.szPassword, \
                addr szPassword
            mov    @stMsg.MsgHead.dwLength, \
                sizeof MSG_HEAD+sizeof MSG_LOGIN
            mov    @stMsg.MsgHead.dwCmdId, CMD_LOGIN
            invoke _SendData, addr @stMsg, \
                @stMsg.MsgHead.dwLength
        .else
            invoke MessageBox, hWinMain, addr szErrConnect, \
                NULL, MB_OK or MB_ICONSTOP
            invoke _Disconnect
        .endif
    .elseif ax == FD_CLOSE
        call    _Disconnect
    .endif
;*****
.elseif eax == WM_COMMAND
    mov    eax, wParam
    .if    ...
        ...
    .elseif ax == IDC_LOGIN
        invoke _Connect
    .elseif ax == IDC_LOGOUT
        invoke _Disconnect
;*****
    .elseif ax == IDOK
        invoke lstrcpy, addr @stMsg.MsgUp.szContent, addr szText
        invoke strlen, addr @stMsg.MsgUp.szContent
        inc    eax
        mov    @stMsg.MsgUp.dwLength, eax
        add    eax, sizeof MSG_HEAD+MSG_UP.szContent
        mov    @stMsg.MsgHead.dwLength, eax
        mov    @stMsg.MsgHead.dwCmdId, CMD_MSG_UP
        invoke _SendData, addr @stMsg, @stMsg.MsgHead.dwLength
        invoke SetDlgItemText, hWinMain, IDC_TEXT, NULL
        invoke GetDlgItem, hWinMain, IDC_TEXT
        invoke SetFocus, eax
    .endif
    ...

```

下面按照连接→发送数据包→接收数据包→断开连接的顺序对代码进行分析。

按下“登录”按钮后，程序在 WM_COMMAND 消息的处理代码中调用_Connect 子程序，子程序中首先将服务器 IP 地址、用户名、密码文本框和“登录”按钮灰化，然后用 socket 函数创建套接字，再用 WSAAsyncSelect 函数将套接字的通知消息绑定到对话框窗口中，注意函数中用到的 WM_SOCKET 消息并不是预定义的消息编号，而是在程序的开始部分自定义为

WM_USER+100 的。

接下来就是用 connect 函数连接到服务器了，非阻塞模式下 connect 函数肯定返回失败，因为连接的过程不可能马上完成，但必须在检测到出错代码不是 WSAEWOULDBLOCK 时才表示真正失败，这时程序显示对应的信息并调用_DisConnect 子程序，_Disconnect 子程序的功能是关闭套接字并且重新激活服务器 IP 地址、用户名、密码文本框和“登录”按钮，以便进行下一次登录操作。如果调用 connect 函数得到的出错代码是 WSAEWOULDBLOCK，子程序只需直接返回即可。

等 connect 函数真正执行完毕后，窗口过程会被调用，对应的消息是包含 FD_CONNECT 通知码的 WM_SOCKET 消息，在相应的处理代码中，如果发现 lParam 的高 16 位不为 0，程序同样提示“无法连接到服务器”并调用_DisConnect 子程序，lParam 的高 16 位为 0 则表示连接成功，这时应该向服务器端发送登录数据包了。

但是发送数据包的操作却有点难度，因为非阻塞模式下，send 函数实际发送的字节数可能少于请求发送的字节数，而且发送中可能会遇到发送缓冲区满的情况，这时程序不能机械地等待发送缓冲区变空，而必须退出窗口过程，否则可能会影响其他消息的处理，只有等下次收到 FD_WRITE 通知码后才能继续发送未发送的部分。

为了处理这些情况，程序定义了一个足够大的发送缓冲区 szSendMsg（例子中定为 10 个 MSG_STRUCT 结构的长度），如果需要发送数据，程序将数据首先移入发送缓冲区并进行发送，在发送的过程中，每次调用 send 函数后，将缓冲区头部已发送的部分丢弃并把后续的数据前移，如果调用 send 函数没有得到 WSAEWOULDBLOCK 错误，则循环将发送缓冲区发完为止，万一中途得到 WSAEWOULDBLOCK 错误，则灰化聊天语句输入框和“发送”按钮并退出（这是为了防止用户继续发送聊天语句）。退出时可能还有部分数据未发送完毕，没关系，因为数据保存在全局的数据段中，不会随退出而丢失。

一旦窗口过程收到 FD_WRITE 通知码，那么程序重新激活聊天语句输入框和“发送”按钮，并继续用上述算法重新发送缓冲区中的未完成数据，完成上述发送功能的代码包含在_SendData 子程序中，子程序的两个参数为要发送的数据地址和数据长度，如果调用的时候两个参数指定为 0，则表示没有新的数据要发送，程序将仅发送缓冲区中的未发送部分，读者可以看到，收到 FD_WRITE 通知时两个参数指定为 0，而在“发送”按钮的响应代码中，两个参数指定为 MSG_UP 数据包的地址和长度。

接收数据包的时候也需要多次拼接，比如，得到 FD_READ 通知码后去接收数据时，收到的数据不一定够一个完整的数据包，这时程序不能在 WM_SOCKET 消息中等待，否则会阻塞对其他窗口消息的处理，而是要将收到的数据保存起来，等下次收到 FD_READ 通知码后继续读取，并将多次读取的数据拼接成一个完整的数据包后再进行处理。

接收及拼接数据包的代码在_RecvData 子程序中完成，程序首先在数据段中定义了一个缓冲区 szRecvMsg 用于保存数据，每次得到 FD_READ 通知码后，程序调用_RecvData 子程序，子程序的开始部分首先检查缓冲区中的已有数据长度 dwRecvBufSize，如果长度小于一个数据包头的长度，则下次调用 recv 时接收的字节数为 sizeof MSG_HEAD-dwRecvBufSize；否则，则从

已完整接收的数据包头中取出数据包长度 `dwLength`，并指定 `recv` 函数接收 `dwLength-dwRecvBufSize` 字节的数据。这种算法是为了首先接收完整的数据包头，从数据包头中得到数据包体的长度后再接收数据包体。

每次调用 `recv` 函数后，程序检测缓冲区中的数据长度是否已经是完整的数据包，如果是，则调用 `_ProcMessage` 子程序处理数据包，处理完毕后，程序将缓冲区中的已处理数据包丢弃，也就是将已有数据的计数变量 `dwRecvBufSize` 设置为 0。

现在来关心一下处理数据包的 `_ProcMessage` 子程序。总的来说，这个子程序的处理流程和阻塞模式下的处理流程一致，惟一的不同在于多了一个对 `dbStep` 变量的控制，这个变量是干什么的呢？

在阻塞模式下，程序是按顺序执行的，假如程序逻辑规定第一步是登录，第二步是发送聊天语句，那么程序在第二步执行时，我们就可以确定第一步已经执行过了。但非阻塞模式则不同，不管是哪一步，程序总是在同样的窗口过程中执行（这就是消息驱动方式的坏处，不是吗？），所以需要定义一个 `dbStep` 变量来记录程序的逻辑状态。

与阻塞模式的程序相比，非阻塞模式的缺点之一在于难以维护程序的状态，如果程序中需要同时操作多个套接字，则需要在窗口过程中处理混杂在一起的全局数据，在数据的维护上非常麻烦；缺点之二在于窗口消息的处理方式本质上是串行的，也就是说，其他窗口消息被处理时，就无法处理 `WM_SOCKET` 消息，反之亦然。在实际使用的时候具体使用哪种模式，读者可以根据程序的特点进行选择。

最后给读者留一个思考题：程序中没有链路空闲 60 秒后自动断开的代码，如果要加上这个功能，该如何修改代码呢？

答案是，由于程序中没有一个专门的线程来收发数据包，所以超时检测机制需要在主线程中完成，最适合的办法就是用定时器。程序可以创建一个以 10 秒钟为周期的定时器，并定义一个 `dwLastTime` 变量，在每次成功调用 `recv` 和 `send` 函数后更新 `dwLastTime` 变量，在定时器消息里面判断当前时间和 `dwLastTime` 变量的差值大于 60 秒则调用 `_Disconnect` 子程序即可，定时器在连接成功后创建，在连接断开的时候销毁。



Win32 API 函数的名称一般由几个单词组成，每个单词的首字母是大写的，但是大部分 WinSock 函数的命名却是全部小写的，如 `socket`、`closesocket`、`ntohl` 等，造成这种现象的原因是这些函数名称源于 UNIX socket，而 UNIX socket 中的函数命名全部是小写的。读者也可以看到：WinSock 接口中由 Windows 系统扩展的函数使用的就是标准的 Win32 API 命名方式，如 `WSAStartup` 和 `WSACleanup` 等。从这里也可以看出哪些函数是 WinSock 接口特有的。

16.3.5 其他常用函数

在本章的最后，将介绍一些相对独立的 WinSock 接口函数，这些函数用于实现一些辅助的功能，在前面的例子中，从简化程序、便于理解的角度考虑，没有把这些函数用上去，但在实际的编程中它们还是很常用的。

1. 和主机名相关的函数

在客户端例子代码中，我们在服务器 IP 地址一栏中输入的是类似于 aa.bb.cc.dd 类型的 IP 地址字符串，如果输入类似于 www.sina.com.cn 之类的主机名就不行了，但是在实际的应用中，我们得到的服务器地址经常是一个主机名，要想程序的兼容性好一点，就必须既能处理 IP 地址，也能处理主机名。

和主机名相关的函数有 `gethostbyname`、`gethostbyaddr` 和 `gethostname`。其中 `gethostbyname` 函数可以将主机名转换成 IP 地址。在附书光盘的 Chapter16 目录下有一个 Ping.exe 程序，在命令行中运行该程序（注意，不要运行操作系统自带的 Ping 程序），例如 Ping www.sina.com.cn 的时候，程序的运行结果是：

```
The host [www.sina.com.cn] has 4 IP addresses:
211.95.77.1 / 211.95.77.4 / 211.95.77.3 / 211.95.77.2
Ping first IP 211.95.77.1 with 32 bytes of data:

Reply from 211.95.77.1: bytes=32 time=230ms TTL=55
Reply from 211.95.77.1: bytes=32 time=231ms TTL=55
```

可以发现，程序将 www.sina.com.cn 解析成了 4 个 IP 地址，这个从主机名到 IP 的解析功能就是用 `gethostbyname` 函数完成的。该函数的用法是：

```
invoke    gethostbyname, lphostname
```

函数惟一输入的参数是需要解析的主机名字符串，如果解析失败将返回 0；成功的话，因为一个主机名可能对应多个 IP 地址，所以函数无法直接用 `eax` 返回所有的 IP 地址，函数返回的是一个指针，指向位于 WinSock 接口内部缓冲区中的一个 `hostent` 结构中，这个结构的定义是：

```
hostent STRUCT
    h_name      DWORD    ?           ;指针，指向和 IP 地址对应的主机名
    h_alias     DWORD    ?           ;指针，指向一个包含别名指针的列表
    h_addr      WORD     ?           ;返回的 IP 地址类型
    h_len       WORD     ?           ;每个地址的长度
    h_list      DWORD    ?           ;指向一个指针列表
hostent ENDS
```

目标主机的 IP 地址由 `h_list` 字段来返回，如图 16.9 所示，`gethostbyname` 函数返回 `hostent` 结构指针，`hostent` 结构中的 `h_list` 字段本身也是一个指针，它指向一个 IP 地址的指针列表，当主机名对应多个 IP 地址的时候，这个指针列表中就存在多个表项，最后一个表项总是 NULL，用来指示列表的结束。真正的 IP 地址数据的存放位置由指针列表中的多个指针指出。

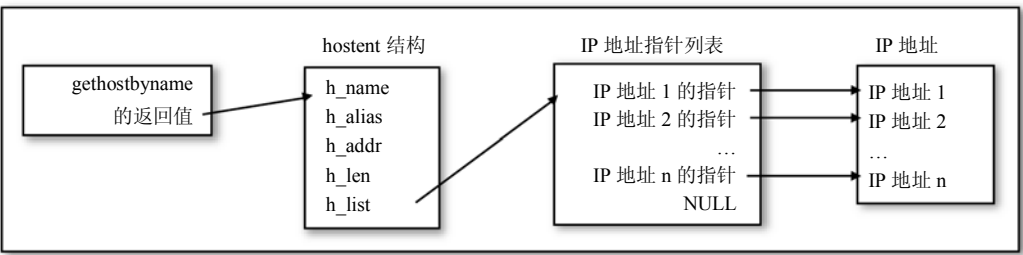


图 16.9 从 gethostbyname 函数的返回值得到 IP 地址

所以对 gethostbyname 函数的返回值要经过多次指针的转换后才能得到 IP 地址数据，比较麻烦，如果要得到所有 IP 地址的话，处理代码如下：

```
invoke    gethostbyname, addr szHostName
.if       eax
    mov     eax, [eax + hostent.h_list] ;取 h_list 指针
    .while  dword ptr [eax]
        mov     ecx, [eax]           ;取一个 IP 地址的指针
        mov     ecx, [ecx]           ;用指针取出 IP 地址
        ;现在 ecx 中就是 IP 地址, 可以进行处理了!
        add     eax, 4                ;指向下一个 IP 地址指针
    .endw
.endif
```

如果只需要得到第一个 IP 地址，那么就不需要循环了：

```
invoke    gethostbyname, addr szHostName
.if       eax
    mov     eax, [eax + hostent.h_list] ;取 h_list 指针
    mov     eax, [eax]
    mov     eax, [eax]
.endif
; 现在 eax 中就是第一个 IP 地址
```

函数返回的 IP 地址已经是网络字节顺序的，可以直接用在其他函数中。

如果输入的是一个类似于 aa.bb.cc.dd 类型的 IP 地址字符串，函数也能将其转换成正确的 IP 地址，但是 gethostbyname 函数是利用 DNS 来解析主机地址的，访问 DNS 服务器需要一些时间。当输入的是 IP 地址字符串时，虽然在本地计算一下就可以得到 IP 地址，但函数还是会机械地去访问 DNS 服务器，造成不必要的延时。

而用 inet_addr 函数来转换 IP 地址字符串的工作是在本地运算的，不会有延时，所以在实际使用中，一般先尝试用 inet_addr 函数将输入的地址字符串当成 IP 地址串来转换，如果不成功的话，再当做主机名来解析，这样就不会造成不必要的等待。

具体的代码见下面的_GetHostIp 子程序，子程序的输入参数是指向 IP 地址字符串或者主机名字符串的指针，如果转换成功，则返回 IP 地址，否则返回 0：

```
_GetHostIp    proc        uses ecx edx _lpHostName

    invoke     inet_addr, _lpHostName    ;首先当做 IP 地址字符串处理
    .if       eax == INADDR_NONE        ;如果不成功则当做主机名处理
```



```

        invoke    gethostbyname, _lpHostName
        .if      eax
            mov     eax, [eax + hostent.h_list]
            mov     eax, [eax]
            mov     eax, [eax]
        .endif
    .endif
    ret

```

```

_GetHostIp    endp

```

在前面的 Client1 例子中, 只要将 inet_addr 函数的调用语句换成对 _GetHostIp 子程序的调用, 那么在服务器地址输入栏中既可以使用 IP 地址, 也可以使用主机名。读者也可以把这个子程序直接用在自己的程序中。

gethostbyaddr 函数则用来从将 IP 地址转换成主机名, 函数的用法是:

```

invoke    gethostbyaddr, addr dwIP, length, type

```

函数的第一个参数是指向 IP 地址的指针, 注意这里的 IP 地址不是指字符串, 而是指按网络字节顺序排列的 32 位 IP 地址, 第二个参数是前面参数中 IP 地址数据的长度 (当然就是 4 了), 第三个参数是地址的类型, 一般指定为 AF_INET。

如果转换失败, 函数返回 0, 成功的话函数的返回值也是一个指向 hostent 结构的指针, 结构的第一个字段 h_name 指向转换后的主机名。gethostbyaddr 的具体使用例子如下, 这段代码将 dwIP 中的 IP 地址转换成主机名放到 szHostName 中:

```

dwIP          dd      ?
szHostName     db      MAX_PATH dup (?)
...
invoke    gethostbyaddr, addr dwIP, 4, AF_INET
.if      eax
    mov     eax, [eax]           ;得到 h_name 字段
    .if     eax                 ;h_name 是一个字符串指针
        invoke    lstrcpy, addr szHostName, eax
    .endif
.endif

```

最后, 使用 gethostname 函数可以获取本地计算机的主机名:

```

invoke    gethostname, lpbuffer, size

```

这个函数的使用非常简单, 函数将在 lpbuffer 指定的缓冲区中返回本地计算机的主机名字符串, size 参数指定缓冲区的大小。

2. 获取套接字两端的地址信息

当一个 TCP 套接字已经在连接状态的时候, 它的对端使用哪个 IP 地址和端口, 本地又是使用哪个地址和端口呢? 这两种信息可以通过 getpeername 和 getsockname 函数来获取。

getpeername 函数的用法是:

```

invoke    getpeername, s, lpsockaddr, size

```

第一个参数是套接字句柄，`lpsockaddr` 参数指向一个缓冲区，函数会在这里返回一个描述对端使用的 IP 地址和端口的 `sockaddr_in` 结构，`size` 参数指定缓冲区的长度。如果地址信息获取成功，函数返回 0；当套接字未在连接状态，或者其他原因导致无法获取地址信息，则函数返回 `SOCKET_ERROR`。

调用 `getpeername` 函数不是获取对端 IP 地址和端口的惟一办法，实际上，如果连接是本地主动发起的，那么发起连接的时候我们就已经知道对端的 IP 地址和端口，根本不必重新去获取一遍；连接是对方发起的时候，本地程序在调用 `accept` 函数的时候也可以得到对方的地址信息，这一点在 `accept` 函数的介绍中已经说明过了。

`getsockname` 函数则用于获取本地端使用的 IP 地址和端口，读者可能会说，这个 IP 地址就是本机地址呀，还用检测吗？但是当本机有多个 IP 地址的时候，TCP 连接使用的是其中的一个 IP 地址而不是全部 IP 地址，所以通过这个函数可以得知连接究竟是建立在哪个 IP 地址之上的。另外，当本机主动发起连接的时候，我们一般不会主动将套接字 `bind` 到一个特定的端口上，而是让系统自动选择端口，这时可以通过 `getsockname` 函数得知系统选择的是哪个端口号。

`getsockname` 函数的参数和 `getpeername` 一模一样，只不过在缓冲区中返回的是包含本地 IP 地址和端口的 `sockaddr_in` 结构。

第 17 章

PE 文 件

PE 格式是 Windows 下最常用的可执行文件格式，有些应用必须建立在了解 PE 文件的基础上，如可执行文件的加密解密、文件型病毒的查杀，等等。本章将具体讨论与 PE 文件相关的内容。

与 PE 文件格式相关的资料可以说是不少，但大多数只是列举了 PE 文件头中一些数据结构和字段含义的介绍，阅读后让读者很难有全局的概念，为了避免这种做法，本章将以一些 PE 文件的操作例子来举例说明如何对 PE 文件进行编程。

17.1 PE 文件的结构

17.1.1 概论

在一个操作系统中，可执行的代码在被最终装入内存执行之前是以文件的方式存放在磁盘中的，DOS 操作系统中的 COM 文件是最早的也是结构最简单的可执行文件，COM 文件中仅仅包括可执行代码，没有附带任何“支持性”的数据，所以 COM 文件在使用方便的同时也存在诸多限制：首先是没有附加数据来指定文件入口，这样，第一句执行指令必须安排在文件头部；再就是没有重定位信息，这样代码中不能有跨段操作数据的指令，造成代码和数据，甚至包括堆栈只能限制在同一个 64 KB 的段中。

为了更灵活地使用可执行代码，DOS 系统中又定义了另一种可执行文件，那就是我们熟悉的 EXE 文件，EXE 文件在代码的前面加了一个文件头，文件头中包括各种说明数据，如文件入口、堆栈的位置、重定位表等等，操作系统根据文件头中的信息将代码部分装入内存，根据重定位表修正代码，最后在设置好堆栈后从文件头中指定的入口开始执行。

显然，可执行文件的格式是操作系统工作方式的写照，因为可执行文件头部的数据是供操作系统装载文件用的，不同操作系统的运行方式各不相同，所以造成可执行文件的格式各不相同。

当 Windows 3.x 出现的时候，可执行文件中出现了 32 位代码，程序运行时转到保护模式之前需要在实模式下做一些初始化，这样实模式的 16 位代码必须和 32 位代码一起放在可执行

文件中，旧的 DOS 可执行文件格式无法满足这个要求，所以 Windows 3.x 执行文件使用新的 LE 格式的可执行文件（Linear executable/线性可执行文件），Windows 9x 中的 VxD 驱动程序也使用 LE 格式，因为这些驱动程序中也同时包括 16 位和 32 位代码。

而在 Windows 9x, Windows NT, Windows 2000 下，纯 32 位的可执行文件都使用微软设计的一种新的文件格式——PE 格式（Portable Executable File Format/可移植的执行体）。

PE 文件的基本结构如图 17.1 所示，在 PE 文件中，代码、已初始化的数据、资源和重定位信息等数据被按照属性分类放到不同的 Section（节区/或简称为节）中，而每个节区的属性和位置等信息用一个 IMAGE_SECTION_HEADER 结构来描述，所有的 IMAGE_SECTION_HEADER 结构组成一个节表（Section Table），节表数据在 PE 文件中被放在所有节数据的前面。我们知道，Win32 中可以对每个内存页分别指定可执行、可读写等属性，在 PE 文件中将同样属性的数据分类放在一起是为了统一描述这些数据装入内存后的页面属性。

由于数据是按照属性在节中放置的，不同用途但是属性相同的数据（如导入表、导出表以及 .const 段指定的只读数据）可能被放在同一个节中，所以 PE 文件中还用一系列的数据目录结构 IMAGE_DATA_DIRECTORY 来分别指明这些数据的位置，数据目录表和其他描述文件属性的数据合在一起称为 PE 文件头，PE 文件头被放置在节和节表的前面。

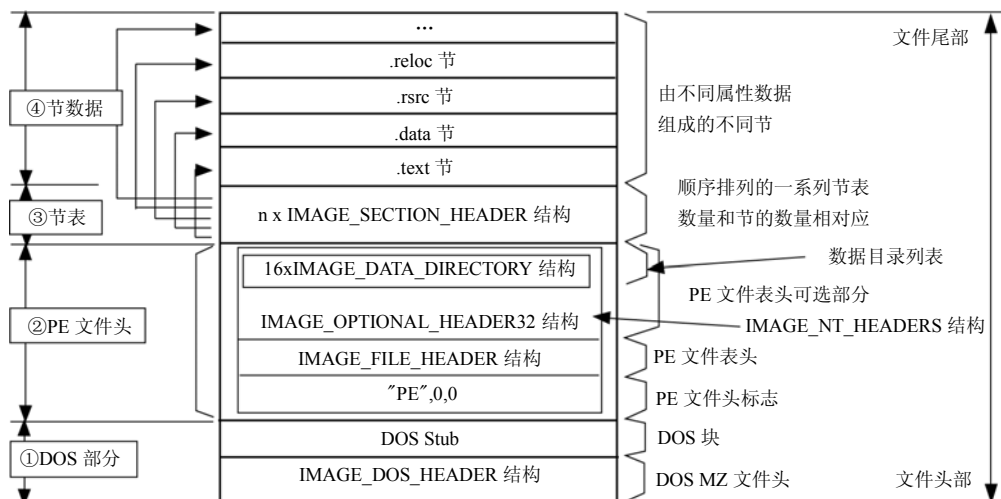


图 17.1 PE 文件的基本结构

上面介绍的这些部分是 PE 文件中真正用于 Win32 的部分，为了与 DOS 系统的文件格式兼容，在这部分的前面又加上了一个标准的 DOS MZ 格式的可执行部分，所有这些部分合起来组成了现在使用的 PE 文件。下面分别介绍这些组成部分。

17.1.2 DOS 文件头和 DOS 块

PE 文件中还包括一个标准的 DOS 可执行文件部分，如图 17.1 中左边的①所示，这看上去有些奇怪，但是这对于可执行文件的向下兼容性来说却是不可缺少的。

操作系统识别可执行文件的方法是按照文件格式而不是按照扩展名，所以，虽然 DOS 的传统 EXE 文件、LE 格式和 PE 格式的可执行文件都沿用了 .exe 的扩展名，但是操作系统总是能够正确识别这些文件并按照正确的方法装入它们。如果文件头中的数据格式不符合任何已经定义的格式，那么系统按照 COM 文件的格式装入文件，也就是说将整个文件的数据全部当做代码装入执行。

这个规则说明了为什么很多非 .exe 扩展名的可执行文件（如 LE 格式的 VxD 文件、PE 格式的 .dll, .scr 文件，等等）也能够被装入并正确运行，也说明了为什么把可执行文件的扩展名随意修改为 .exe、.com 或者 .bat（甚至是 .pif, .scr 或者 .bat），系统也能正确识别并执行的原因。

但是这种方法也存在一个问题，假如一个 PE 格式的可执行文件在 Windows 中执行，那没有任何异常，因为 Windows 能够识别 PE 文件头并正确装入，但如果将 PE 文件放入 DOS 执行，那么 DOS 系统肯定无法识别 PE 文件头，假如 PE 文件的头部不包括一个 DOS 部分的话，那么按照前面介绍的规则，PE 文件头的数据会被 DOS 系统作为代码装入并执行，这种操作几乎可以肯定会让系统立刻挂起。

为了避免这种情况，PE 文件的头部包括了一个标准的 DOS MZ 格式的可执行部分，这样万一在 DOS 下执行一个 PE 文件，系统可以将文件解释为 DOS 下的 .exe 可执行格式，并执行 DOS 部分的代码。

一般来说，DOS 部分的执行代码只是简单地显示一个 “This program cannot be run in DOS mode.” 就退出了，这段简单的代码是编译器自动生成的。



如果对编译器内定的这段简单代码不满意的话，读者可以回忆一下第 2 章 2.2.1 节中介绍 link.exe 参数部分的内容，如果在 link 时使用 /stub:dos_file_name.exe 选项，读者完全可以用一个全功能的 DOS 程序来作为 PE 文件的 DOS 部分。

笔者就见过一个 CD 播放程序，在 DOS 下执行是一个文本界面的播放器，而在 Windows 下执行又是标准的 Windows 界面。我们知道，DOS 和 Windows 下不管是界面还是 CD 操作都是完全不同的概念，它们不可能在同一段代码中完成。实际上，这个程序就是用这种方法插入了一个完全独立的 DOS CD 播放程序。

PE 文件中的 DOS 部分由 MZ 格式的文件头和可执行代码部分组成，可执行代码被称为 “DOS 块”（DOS stub）。MZ 格式的文件头由 IMAGE_DOS_HEADER 结构定义：

IMAGE_DOS_HEADER STRUCT			
e_magic	WORD	?	;DOS 可执行文件标记，为 “MZ”
e_cblp	WORD	?	
e_cp	WORD	?	
e_crlc	WORD	?	
e_cparhdr	WORD	?	
e_minalloc	WORD	?	
e_maxalloc	WORD	?	
e_ss	WORD	?	;DOS 代码的初始化堆栈段
e_sp	WORD	?	;DOS 代码的初始化堆栈指针
e_csum	WORD	?	

e_ip	WORD	?	;DOS 代码的入口 IP
e_cs	WORD	?	;DOS 代码的入口 CS
e_lfarlc	WORD	?	
e_ovno	WORD	?	
e_res	WORD	4 dup(?)	
e_oemid	WORD	?	
e_oeminfo	WORD	?	
e_res2	WORD	10 dup(?)	
e_lfanew	DWORD	?	;指向 PE 文件头
IMAGE_DOS_HEADER ENDS			

DOS 文件头的前面部分并不陌生,第一个字段 e_magic 被定义成字符“MZ”(在 Windows.inc 文件中已经预定义为 IMAGE_DOS_SIGNATURE)作为识别标志,后面的一些字段指明了入口地址、堆栈位置和重定位表位置等。

标准的 DOS 文件头的定义只到 e_ovno 字段位置,后面的这些字段是在 Windows 系统出现后为了定义 LE、PE 等文件格式而扩充的,DOS 系统对这些字段不进行解释。对于 PE 文件来说,有用的是最后的 e_lfanew 字段,这个字段指出了真正的 PE 文件头(如图 17.1 中的②所示)在文件中的位置,这个位置总是以 8 字节为单位对齐的。

实际上,Windows 中使用的其他几种可执行文件格式也是这样引出的,如果是 LE, LX 等格式的文件,那么 e_lfanew 字段指向的位置会是 LE 文件头和 LX 文件头。

17.1.3 PE 文件头 (NT 文件头)

从 DOS 文件头的 e_lfanew 字段(文件头偏移 003ch)得到真正的 PE 文件头位置后,现在来看看它的定义,PE 文件头是由 IMAGE_NT_HEADERS 结构定义的:

IMAGE_NT_HEADERS STRUCT			
Signature	DWORD	?	;PE 文件标识
FileHeader	IMAGE_FILE_HEADER	<>	
OptionalHeader	IMAGE_OPTIONAL_HEADER32	<>	
IMAGE_NT_HEADERS ENDS			

PE 文件头的第一个双字是一个标志,它被定义为 00004550h,也就是字符“P”,“E”加上两个 0,这也是“PE”这个称呼的由来,大部分的文件属性由标志后面的 IMAGE_FILE_HEADER 和 IMAGE_OPTIONAL_HEADER32 结构来定义,从名称看,似乎后面的这个 PE 文件表头结构是可选的(Optional),但实际上这个名称是名不符实的,因为它总是存在于每个 PE 文件中。

1. IMAGE_FILE_HEADER 结构

IMAGE_FILE_HEADER 结构的定义如下所示,字段后面的注释中标出了字段相对于 PE 文件头的偏移量,以供读者快速参考:

IMAGE_FILE_HEADER STRUCT			
Machine	WORD	?	;0004h - 运行平台
NumberOfSections	WORD	?	;0006h - 文件的节数目
TimeDateStamp	DWORD	?	;0008h - 文件创建日期和时间
PointerToSymbolTable	DWORD	?	;000ch - 指向符号表(用于调试)
NumberOfSymbols	DWORD	?	;0010h - 符号表中的符号数量(用于调试)

SizeOfOptionalHeader	WORD	?	;0014h - IMAGE_OPTIONAL_HEADER32 结构的长度
Characteristics	WORD	?	;0016h - 文件属性
IMAGE_FILE_HEADER ENDS			

几个关键字段的含义解释如下。

● Machine 字段

用来指定文件的运行平台，常见的定义值如表 17.1 所示。Windows 可以运行在 Intel 和 Sun 等几种不同的硬件平台上，不同平台指令的机器码是不同的，为不同平台编译的可执行文件显然无法通用。如果 Windows 检测到这个字段指定的适用平台与当前的硬件平台不兼容，它将拒绝装入这个文件。

表 17.1 运行平台识别码的定义（更多定义参见 Windows.inc 文件）

Windows.inc 中的预定义值	十六进制值	说 明
IMAGE_FILE_MACHINE_UNKNOWN	0	未知平台
IMAGE_FILE_MACHINE_I386	014ch	Intel 386
暂无	014dh	Intel 486
暂无	014eh	Intel 586
暂无	0160h	R3000（大尾方式）
IMAGE_FILE_MACHINE_R3000	0162h	R3000（小尾方式）
IMAGE_FILE_MACHINE_R4000	0166h	R4000（小尾方式）
IMAGE_FILE_MACHINE_R10000	0168h	R10000（小尾方式）
IMAGE_FILE_MACHINE_ALPHA	0184h	Dec Alpha AXP
IMAGE_FILE_MACHINE_POWERPC	01f0h	IBM Power PC（小尾方式）
IMAGE_FILE_MACHINE_ALPHA64	0284h	Dec Alpha AXP64

● NumberOfSections 字段

指出文件中存在的节的数量（如图 17.1 中的④所示），同样，节表的数量（如图 17.1 中的③所示）也等于节的数量。

● TimeDateStamp 字段

编译器创建此文件的时间，它的数值是从 1969 年 12 月 31 日下午 4:00 开始到创建时间为止的总秒数。

● PointerToSymbolTable 和 NumberOfSymbols 字段

这两个字段并不重要，它们与调试用的符号表有关。

● SizeOfOptionalHeader 字段

紧接在当前结构下面的 IMAGE_OPTIONAL_HEADER32 结构的长度，这个值等于 00e0h。

● Characteristics 字段

属性标志字段，它的不同数据位定义了不同的文件属性，具体内容如表 17.2 所示，这是一个很重要的字段，不同的定义将影响系统对文件的装入方式，比如，当位 13 为 1 时，表示这是一个 DLL 文件，那么系统将使用调用 DLL 入口函数的方式调用文件入口，否则的话，表示这是一

个普通的可执行文件，系统直接跳到入口处执行。对于普通的可执行 PE 文件，这个字段的值一般是 010fh，而对于 DLL 文件来说，这个字段的值一般是 210eh。

表 17.2 属性位字段的含义

数 据 位	Windows. inc 中的预定义值	为 1 时的含义
0	IMAGE_FILE_RELOCS_STRIPPED	文件中不存在重定位信息
1	IMAGE_FILE_EXECUTABLE_IMAGE	文件是可执行的
2	IMAGE_FILE_LINE_NUMS_STRIPPED	不存在行信息
3	IMAGE_FILE_LOCAL_SYMS_STRIPPED	不存在符号信息
7	IMAGE_FILE_BYTES_REVERSED_LO	小尾方式
8	IMAGE_FILE_32BIT_MACHINE	只在 32 位平台上运行
9	IMAGE_FILE_DEBUG_STRIPPED	不包含调试信息
10	IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	不能从可移动盘（如软盘、光盘）运行
11	IMAGE_FILE_NET_RUN_FROM_SWAP	不能从网络运行
12	IMAGE_FILE_SYSTEM	系统文件（如驱动程序），不能直接运行
13	IMAGE_FILE_DLL	这是一个 DLL 文件
14	IMAGE_FILE_UP_SYSTEM_ONLY	文件不能在多处理器上计算机上运行
15	IMAGE_FILE_BYTES_REVERSED_HI	大尾方式

2. IMAGE_OPTIONAL_HEADER32 结构

定义 IMAGE_OPTIONAL_HEADER32 结构的本意在于让不同的开发者能够在 PE 文件头中使用自定义的数据，这就是结构名称中“Optional”一词的由来，但实际上 IMAGE_FILE_HEADER 结构不足以用来定义 PE 文件的属性，反而在这个“可选”的部分中有着更多的定义数据，对于读者来说，可以完全不必考虑这两个结构的区别在哪里，只要把它们当成是连在一起的“PE 文件头结构”就可以了。

IMAGE_OPTIONAL_HEADER32 结构的定义如下，同样，字段后面的注释中标出了字段本身相对于 PE 文件头的偏移量：

```

IMAGE_OPTIONAL_HEADER32 STRUCT
    Magic                WORD ?      ;0018h 107h=ROM Image, 10Bh=exe Image
    MajorLinkerVersion   BYTE ?      ;001ah 链接器版本号
    MinorLinkerVersion   BYTE ?      ;001bh
    SizeOfCode           DWORD ?     ;001ch 所有含代码的节的总大小
    SizeOfInitializedData DWORD ?     ;0020h 所有含已初始化数据的节的总大小
    SizeOfUninitializedData DWORD ?   ;0024h 所有含未初始化数据的节的大小
    AddressOfEntryPoint  DWORD ?     ;0028h 程序执行入口 RVA
    BaseOfCode           DWORD ?     ;002ch 代码的节的起始 RVA
    BaseOfData           DWORD ?     ;0030h 数据的节的起始 RVA
    ImageBase           DWORD ?     ;0034h 程序的建议装载地址
    SectionAlignment     DWORD ?     ;0038h 内存中的节的对齐粒度
    FileAlignment        DWORD ?     ;003ch 文件中的节的对齐粒度
    MajorOperatingSystemVersion WORD ? ;0040h 操作系统主版本号
    MinorOperatingSystemVersion WORD ? ;0042h 操作系统副版本号
    MajorImageVersion    WORD ?     ;0044h 可运行于操作系统的最小版本号
    MinorImageVersion    WORD ?     ;0046h
    MajorSubsystemVersion WORD ?     ;0048h 可运行于操作系统的最小子版本号
    MinorSubsystemVersion WORD ?     ;004ah

```

Win32VersionValue	DWORD ?	;004ch 未用
SizeOfImage	DWORD ?	;0050h 内存中整个 PE 映像尺寸
SizeOfHeaders	DWORD ?	;0054h 所有头+节表的大小
Checksum	DWORD ?	;0058h
Subsystem	WORD ?	;005ch 文件的子系统
DllCharacteristics	WORD ?	;005eh
SizeOfStackReserve	DWORD ?	;0060h 初始化时的堆栈大小
SizeOfStackCommit	DWORD ?	;0064h 初始化时实际提交的堆栈大小
SizeOfHeapReserve	DWORD ?	;0068h 初始化时保留的堆大小
SizeOfHeapCommit	DWORD ?	;006ch 初始化时实际提交的堆大小
LoaderFlags	DWORD ?	;0070h 未用
NumberOfRvaAndSizes	DWORD ?	;0074h 下面的数据目录结构的数量
DataDirectory	IMAGE_DATA_DIRECTORY 16 dup(<>)	;0078h
IMAGE_OPTIONAL_HEADER32 ENDS		

这个结构中的大部分字段都不重要，读者可以从注释中理解它们的含义，下面说明的这些字段是比较重要的。

- AddressOfEntryPoint 字段

指出文件被执行时的入口地址，这是一个 RVA 地址（RVA 的含义在下一节中详细介绍）。如果在一个可执行文件上附加了一段代码并想让这段代码首先被执行，那么只需要将这个入口地址指向附加的代码就可以了。

- ImageBase 字段

指出文件的优先装入地址。也就是说当文件被执行时，如果可能的话，Windows 优先将文件装入到由 ImageBase 字段指定的地址中，只有指定的地址已经被其他模块使用时，文件才被装入到其他地址中。链接器产生可执行文件的时候对应这个地址来生成机器码，所以当文件被装入这个地址时不需要进行重定位操作，装入的速度最快，如果文件被装载到其他地址的话，将不得不进行重定位操作，这样就要慢一点。

对于 EXE 文件来说，由于每个文件总是使用独立的虚拟地址空间，优先装入地址不可能被其他模块占据，所以 EXE 总是能够按照这个地址装入，这也意味着 EXE 文件不再需要重定位信息。对于 DLL 文件来说，由于多个 DLL 文件全部使用宿主 EXE 文件的地址空间，不能保证优先装入地址没有被其他的 DLL 使用，所以 DLL 文件中必须包含重定位信息以防万一。因此，在前面介绍的 IMAGE_FILE_HEADER 结构的 Characteristics 字段中，DLL 文件对应的 IMAGE_FILE_RELOCS_STRIPPED 位总是为 0，而 EXE 文件的这个标志位总是为 1。

在链接的时候，可以通过对 link.exe 指定 /base:address 选项来自定义优先装入地址，如果不指定这个选项的话，一般 EXE 文件的默认优先装入地址被定为 00400000h，而 DLL 文件的默认优先装入地址被定为 10000000h。

- SectionAlignment 字段和 FileAlignment 字段

SectionAlignment 字段指定了节被装入内存后的对齐单位。也就是说，每个节被装入的地址必定是本字段指定数值的整数倍。而 FileAlignment 字段指定了节存储在磁盘文件中时的对齐单位。

- Subsystem 字段

指定使用界面的子系统，它的取值如表 17.3 所示。这个字段决定了系统如何为程序建立初始的界面，链接时的/subsystem:xxx 选项指定的就是这个字段的值，在前面章节的编程中我们早已知道：如果将子系统指定为 Windows CUI，那么系统会自动为程序创建一个控制台窗口，而指定为 Windows GUI 的话，窗口必须由程序自己创建。

表 17.3 界面子系统的取值和含义

取 值	Windows.inc 中的预定义值	含 义
0	IMAGE_SUBSYSTEM_UNKNOWN	未知的子系统
1	IMAGE_SUBSYSTEM_NATIVE	不需要子系统（如驱动程序）
2	IMAGE_SUBSYSTEM_WINDOWS_GUI	Windows 图形界面
3	IMAGE_SUBSYSTEM_WINDOWS_CUI	Windows 控制台界面
5	IMAGE_SUBSYSTEM_OS2_CUI	OS2 控制台界面
7	IMAGE_SUBSYSTEM_POSIX_CUI	POSIX 控制台界面
8	IMAGE_SUBSYSTEM_NATIVE_WINDOWS	不需要子系统
9	IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	Windows CE 图形界面

• DataDirectory 字段

这个字段可以说是最重要的字段之一，它由 16 个相同的 IMAGE_DATA_DIRECTORY 结构组成，虽然 PE 文件中的数据是按照装入内存后的页属性归类而被放在不同的节中的，但是这些处于各个节中的数据按照用途可以被分为导出表、导入表、资源、重定位表等数据块，这 16 个 IMAGE_DATA_DIRECTORY 结构就是用来定义多种不同用途的数据块的。IMAGE_DATA_DIRECTORY 结构的定义很简单，它仅仅指出了某种数据块的位置和长度。

IMAGE_DATA_DIRECTORY STRUCT			
VirtualAddress	DWORD	?	;数据的起始 RVA
isize	DWORD	?	;数据块的长度
IMAGE_DATA_DIRECTORY ENDS			

如果将这 16 个 IMAGE_DATA_DIRECTORY 结构按照排列顺序编号为索引号 0 到 15，那么其用途和索引号是一一对应的，其对应关系如表 17.4 所示。

表 17.4 数据目录列表的含义

索 引	索引值在 Windows.inc 中的预定义值	对应的数据块
0	IMAGE_DIRECTORY_ENTRY_EXPORT	导出表
1	IMAGE_DIRECTORY_ENTRY_IMPORT	导入表
2	IMAGE_DIRECTORY_ENTRY_RESOURCE	资源
3	IMAGE_DIRECTORY_ENTRY_EXCEPTION	异常（具体资料不详）
4	IMAGE_DIRECTORY_ENTRY_SECURITY	安全（具体资料不详）
5	IMAGE_DIRECTORY_ENTRY_BASERELOC	重定位表
6	IMAGE_DIRECTORY_ENTRY_DEBUG	调试信息

续表

索 引	索引值在 Windows.inc 中的预定义值	对应的数据块
-----	-------------------------	--------

7	IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	版权信息
8	IMAGE_DIRECTORY_ENTRY_GLOBALPTR	具体资料不详
9	IMAGE_DIRECTORY_ENTRY_TLS	Thread Local Storage
10	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	具体资料不详
11	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	具体资料不详
12	IMAGE_DIRECTORY_ENTRY_IAT	导入函数地址表
13	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	具体资料不详
14	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	具体资料不详
15	未使用	

在 PE 文件中寻找特定的数据时就是从这些 IMAGE_DATA_DIRECTORY 结构开始的，比如要存取资源，那么必须从第 3 个 IMAGE_DATA_DIRECTORY 结构（索引为 2）中得到资源数据块的大小和位置；同理，如果要查看 PE 文件导入了哪些 DLL 文件的哪些 API 函数，那就必须首先从第 2 个 IMAGE_DATA_DIRECTORY 结构得到导入表的位置和大小。

17.1.4 节表和节

从排列位置来看，PE 文件在 DOS 部分和 PE 文件头部分以后就是节表和多个不同的节（如图 17.1 中的③和④所示）。要理解什么是节表，什么是节以及它们之间的关系，那就首先要了解 Windows 是如何将 PE 文件映射到内存的。

1. PE 文件到内存的映射

在执行一个 PE 文件的时候，Windows 并不在一开始就将整个文件读入内存，而是采用与内存映射文件类似的机制，也就是说，Windows 装载器在装载的时候仅仅建立好虚拟地址和 PE 文件之间的映射关系，只有真正执行到某个内存页中的指令或者访问某一页中的数据时，这个页面才会被从磁盘提交到物理内存，这种机制使文件装入的速度和文件大小没有太大的关系。

但是系统装载可执行文件的方法又不完全等同于内存映射文件。当使用内存映射文件时，系统对“原著”非常忠实，如果将磁盘文件和内存映像对比一下，可以发现不管是数据本身还是数据之间的相对位置都是完全相同的。而装载可执行文件的时候，有些数据在装入前会被预先处理（如需要重定位的代码），而装入以后，数据之间的相对位置也可能改变，如图 17.2 所示，一个节的偏移和大小在装入内存前后可能是完全不同的。

Windows 装载器在装载 DOS 部分、PE 文件头部分和节表部分时不进行任何处理，而装载节的时候将根据节的属性做不同的处理，一般需要处理以下几个方面的内容。

● 内存页的属性

对于磁盘映射文件来说，所有的页都是按照磁盘映射文件函数指定的属性设置的，但是装载可执行文件时，与节对应的内存页的属性要按照节的属性来设置。所以在同属一个模块的内存页中，从不同节映射过来的内存页的属性是不同的。

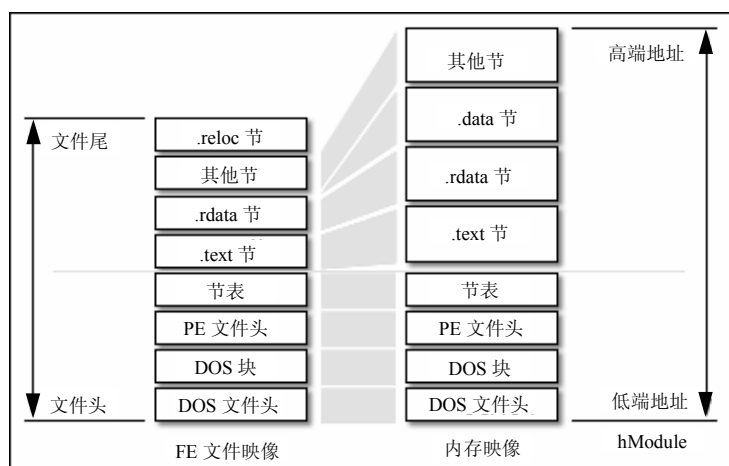


图 17.2 PE 文件到内存的映射

● 节的偏移地址

节的起始地址在磁盘文件中是按照 IMAGE_OPTIONAL_HEADER32 结构的 FileAlignment 字段的值对齐的，而被装载到内存中时是按照同一结构中的 SectionAlignment 字段的值对齐的，两者的值可能不同，所以一个节被装入内存后相对于文件头的偏移和在磁盘文件中的偏移可能是不同的。

节是相同属性数据的组合，当节被装入到内存中的时候，同一个节对应的内存页将被赋予相同的页属性，Windows 系统对内存属性的设置是以页为单位来进行的，所以节在内存中的对齐单位必须至少是一个页的大小，对于 Win32 来说，这个值是 4 KB (1000h)，而对于 Win64 来说，这个值是 8 KB (2000h)。节在磁盘文件中的对齐单位就没有最小 4 KB 的限制，为了减少磁盘文件的大小，文件对齐的单位一般要小于内存对齐的单位（FileAlignment 的值一般为 200h），这样，在磁盘中就不必为每个节最后的零头数据补足 4 KB 的大小了。

● 节的尺寸

对节尺寸的处理有两个方面，首先是由于磁盘映像和内存映像中节对齐单位的不同而造成的长度扩展；其次是对包含未初始化数据的节（如图 17.2 中的 .data 节）。

对于未初始化数据来说（比如在源代码中定义的 .data 段），没必要为它们在磁盘文件中预留空间，只要在可执行文件被装载到内存中后为它们分配空间就可以了，所以包含未初始化数据的节在磁盘文件中的长度被定义为 0，但是被装载到内存中的地址和大小是被明确指定的。对于这种节来说，它所包含的内存页并没有磁盘文件内容与之对应，这些内存页是 Windows 装载器根据节的定义额外开辟出来的。

● 不进行映射的节

有些节中包含的数据仅仅在装载的时候用到，当文件装载完毕的时候，它们不会被递交到物理内存页。最典型的例子就是包含重定位数据的节（如图 17.2 中的 .reloc 节），重定位数据对于文件的执行代码来说是透明的，它只供 Windows 装载器使用，执行代码根本不会去访问

它们，一旦装载完毕，继续为它们提交内存页是一种浪费。所以这些节存在于磁盘文件中，但并不会被映射到内存中。

2. 节表

PE 文件中所有节的属性都被定义在节表中，节表由一系列的 `IMAGE_SECTION_HEADER` 结构排列而成，每个结构用来描述一个节，结构的排列顺序和它们描述的节在文件中的排列顺序是一致的。节表总是被存放在紧接在 PE 文件头的地方，也就是从 PE 文件头（注意：不是文件本身的头部）开始的偏移为 `00f 8h` 的地方。

节表中 `IMAGE_SECTION_HEADER` 结构的总数由 PE 文件头 `IMAGE_NT_HEADERS` 结构中的 `FileHeader.NumberOfSections` 字段指定。

`IMAGE_SECTION_HEADER` 结构的定义如下：

```

IMAGE_SECTION_HEADER STRUCT
    Name1 db IMAGE_SIZEOF_SHORT_NAME dup(?)    ;8 个字节的节区名称
    union Misc
        PhysicalAddress dd ?
        VirtualSize dd ?                        ;节区的尺寸
    ends
    VirtualAddress dd ?                          ;节区的 RVA 地址
    SizeOfRawData dd ?                          ;在文件中对齐后的尺寸
    PointerToRawData dd ?                       ;在文件中的偏移
    PointerToRelocations dd ?                   ;在 OBJ 文件中使用
    PointerToLinenumbers dd ?                   ;行号表的位置（供调试用）
    NumberOfRelocations dw ?                    ;在 OBJ 文件中使用
    NumberOfLinenumbers dw ?                    ;行号表中行号的数量
    Characteristics dd ?                       ;节的属性
IMAGE_SECTION_HEADER ENDS

```

结构中的有些字段是供 COFF 格式的 obj 文件使用的，对可执行文件来说不代表任何意义，在分析的时候可以不予理会，真正有用的几个字段说明如下。

● Name1 字段

这个字段的字段名原来应该是“Name”，但是这个名称和 MASM 中的关键字冲突，所以在定义的时候改为“Name1”，Name1 字段定义了节的名称，字段的长度为 8 个字节。

PE 文件中的节的名称是一个由 ANSI 字符组成的字符串，但并没有规定以 0 结束，如果节的名称字符串长度小于 8 个字节的话，后面以 0 补齐，但是字符串长度达到 8 个字节的话，后面就没有 0 字符了，所以在处理的时候要注意字符串的结束方式。

每个节的名称是惟一的，不能有同名的两个节，但是节的名称不代表任何含义，它仅仅是为了查看方便而设置的一个标记而已，可以选择任何名称甚至将它空着也可以，将包含代码的节命名为“DATA”或者将包含数据的节命名为“CODE”都是合法的。

各种编译器都以自己的方式对节进行命名，所以，在 PE 文件中可以看到各式各样的节名称，比如，在 MASM32 产生的可执行文件中，代码节被命名为“.text”；可读写的数节被命名为“.data”；包含只读数据、导入表以及导出表的节被命名为“.rdata”；而资源节被命

名为“.rsrc”等。但是在其他一些编译器中，导入表被单独放在“.idata”中；而代码节可能被命名为“.code”。



当从 PE 文件中读取需要的节时，不能以节的名称作为定位标准，正确的方法是按照 IMAGE_OPTIONAL_HEADER32 结构中的数据目录字段定位。

笔者曾看过一篇介绍如何存取 PE 文件资源的文章，其中用查找“.rsrc”节的方法得到资源，虽然大部分情况下用这种方法也可以正确地找到资源，但是严格地讲，只有数据目录的 IMAGE_DIRECTORY_ENTRY_RESOURCE 项才永远正确地指向资源数据。

- VirtualSize 字段

代表节的大小，这是节的数据在没有进行对齐处理前的实际大小。

- VirtualAddress 字段

指出节被装载到内存中后的偏移地址，这是一个 RVA 地址。这个地址是按照内存页对齐的，它的数值总是 SectionAlignment 的值的整数倍。

- PointerToRawData 字段

指出节在磁盘文件中的所处的位置。这个数值是从文件头开始算起的偏移量。

- SizeOfRawData 字段

指出节在磁盘文件中所占的空间大小，这个数值等于 VirtualSize 字段的值按照 FileAlignment 的值对齐以后的大小。

依靠这 4 个字段的值，装载器就可以从 PE 文件中找出某个节（从 PointerToRawData 偏移开始的 SizeOfRawData 字节）的数据，并将它映射到内存中去（映射到从模块基地址开始偏移 VirtualAddress 的地方，并占用以 VirtualSize 的值按照页的尺寸对齐后的空间大小）。

- Characteristics 字段

这是节的属性标志字段，其中的不同数据位代表了不同的属性，具体的定义如表 17.5 所示，这些数据位组合起来描述了节的属性。

表 17.5 节的属性标志位含义

位	数据位在 Windows.inc 中的预定义值以及为 1 时的含义
5	(IMAGE_SCN_CNT_CODE 或 00000020h) 节中包含代码
6	(IMAGE_SCN_CNT_INITIALIZED_DATA 或 00000040h) 节中包含已初始化数据
7	(IMAGE_SCN_CNT_UNINITIALIZED_DATA 或 00000080h) 节中包含未初始化数据
25	(IMAGE_SCN_MEM_DISCARDABLE 或 02000000h) 节中的数据在进程开始以后将被丢弃，前面举例的包含重定位表的 .reloc 节就是一个例子
26	(IMAGE_SCN_MEM_NOT_CACHED 或 04000000h) 节中的数据不会经过缓存
27	(IMAGE_SCN_MEM_NOT_PAGED 或 08000000h) 节中的数据不会被交换到磁盘
28	(IMAGE_SCN_MEM_SHARED 或 10000000h) 表示节中的数据将被不同的进程所共享，在第 11 章的钩子例子中的共享数据的节就设置了这个属性标志

续表

位	数据位在 Windows.inc 中的预定义值以及为 1 时的含义
29	(IMAGE_SCN_MEM_EXECUTE 或 20000000h)映射到内存后的页面包含可执行属性
30	(IMAGE_SCN_MEM_READ 或 40000000h)映射到内存后的页面包含可读属性
31	(IMAGE_SCN_MEM_WRITE 或 80000000h)映射到内存后的页面包含可写属性

代码节的属性一般为 60000020h，也就是可执行、可读和“节中包含代码”；数据节的属性一般为 c0000040h，也就是可读、可写和“包含已初始化数据”；而常量节（对应源代码中的 .const 段）的属性为 40000040h，也就是可读和“包含已初始化数据”；资源节的属性和常量节的属性一般是相同的。

当然节属性的定义不一定就是这些值，比如，当 PE 文件被压缩工具压缩以后，包含代码的节往往被同时设置了可执行、可读和可写属性，因为解压部分需要将解压后的代码回写到代码段中。读者可以做个实验：在程序中往代码段写数据，编译链接完成后执行一下肯定会引发异常，然而用 Upx 等压缩软件压缩后再执行，就会发现文件可以正常执行了，这就是因为压缩软件为了解压的需要而将节的属性设置为可写了。

3. RVA 和文件偏移的转换

在前面的内容中已经多次提到“RVA”一词，对于初次接触 PE 文件的读者来说，RVA 是个比较费解的概念，特别是在一开始就去接触 RVA 的情况下。RVA 的概念是和 PE 文件从磁盘到内存的映射息息相关的，在了解了这方面的内容后，再来看 RVA 就不成问题了。

RVA 是相对虚拟地址（Relative Virtual Address）的缩写，顾名思义，它是一个“相对”地址，也可以说是“偏移量”，PE 文件的各种数据结构中涉及地址的字段大部分都是以 RVA 表示的。

准确地说，RVA 就是当 PE 文件被装载到内存中后，某个数据的位置相对于文件头的偏移量。举个例子，如果 Windows 装载器将一个 PE 文件装入 00400000h 处的内存中，而某个节中的某个数据被装入 0040xxxxh 处，那么这个数据的 RVA 就是 (0040xxxxh - 00400000h) = xxxxxh，反过来说，将 RVA 的值加上文件被装载的基地址，就可以找到数据在内存中的实际地址。

很明显，PE 文件中出现 RVA 的概念是因为 PE 的内存映像和磁盘文件映像是不同的，如图 17.3 中的 A 和 A' 所示，同一数据相对于文件头的偏移量在内存中和在磁盘文件中可能是不同的，为了提高效率，PE 文件头中使用的都是内存映像中的偏移量，也就是 RVA。从图 17.3 中也可以得到另一个结论，那就是 RVA 仅仅是对于处于节中的数据而言的，对于文件头和节表来说无所谓 RVA 和文件偏移，因为它们在被映射到内存中后不管是大小还是偏移都不会有任何改变。

使用 RVA，使文件装入内存后的数据定位变得方便，然而却给处理磁盘上的静态 PE 文件带来了很大的麻烦，举例来说，假如要读取 PE 文件中的资源（如图 17.3 中的 A 所示），第一个步骤就是从 PE 文件头的数据目录中访问第 3 个 IMAGE_DATA_DIRECTORY 结构，并从结构中得到资源所处的偏移量，但是这样得到的偏移量是个 RVA，它只能用于在内存中查找由 A' 位置所指示的资源。用它直接在磁盘文件中定位 A 位置是错误的。



(3) 在节表中获取节在文件中所处的偏移 (PointerToRawData 字段)，将这个偏移值加上上一步得到的 RVA' 值，这才是数据在文件中的真正偏移位置。

这里是两个通用的函数，其中 `_RVAToOffset` 函数将 RVA 转换成文件偏移，输入的参数是已经读取到内存中的文件头的地址和 RVA 的值；`_GetRVASection` 函数用来获取 RVA 所在的节的名称。这两个函数被保存在 `_RvaToFileOffset.asm` 文件中，并将在以后的例子中用到，函数中使用的算法就是上面 3 个步骤中列出的算法。

634

@@:

[illegible]

4. PEInfo 例子

好了，到现在为止，相信读者对 PE 文件的结构应该有初步的了解了，现在尝试写一个程序来查看 PE 文件的一些信息，并列出所有节的名称和属性。例子代码在本书所附光盘的 Chapter17\PeInfo 目录中，其中包括资源文件 Main.rc，界面源代码 Main.asm 和处理分析 PE 文件的源代码 _ProcessPeFile.asm。为了节省篇幅，本章以后的几个例子中都将使用同样的界面文件 Main.asm 和 Main.rc，只是处理 PE 文件的代码 ProcessPeFile.asm 有所不同。

通用的界面资源文件 Main.rc 定义如下:

[illegible]

哭

The screenshot shows the PEInfo application window titled "PE文件基本信息". It displays the following information:

- 文件名: Z:\Win32asm\Code\Chapter17\PeInfo\Main.exe
- 运行平台: 0x014C
- 节区数量: 4
- 文件标记: 0x010F
- 建议装入地址: 0x00400000

Below this information is a table with the following columns: 节区名称, 节区大小, 虚拟地址, Raw_尺寸, Raw_偏移, 节区属性.

节区名称	节区大小	虚拟地址	Raw_尺寸	Raw_偏移	节区属性
.text	0000045C	00001000	00000600	00000400	60000020
.rdata	000004AE	00002000	00000600	00000A00	40000040
.data	00000114	00003000	00000000	00000000	C0000040
.rsrsc	00000EAO	00004000	00001000	00001000	40000040

节区名称	节区大小	虚拟地址	Raw_尺寸	Raw_偏移	节区属性
.text	0000045C	00001000	00000800	00000040	60000020
.rdata	000004AE	00002000	00000600	00000A00	40000040
.data	00000114	00003000	00000000	00000000	C0000040
.rsrc	000002A0	00004000	00001000	00001000	40000040

[illegible]

638

639

640

源代码中没有任何新的概念：程序首先创建一个对话框，并在初始化的时候将 RichEdit 控件的字体设置为“宋体”。当用户选择“打开”文件菜单后，则显示一个打开文件通用对话框让用户选择一个文件。

在处理文件之前，程序使用第 14 章中介绍的 SEH 来设置一个异常处理回调函数，一旦发生异常的话，则将程序转移到 `_ErrFormat` 标号处执行并认为文件的格式存在异常。由于 PE 文件的分析中涉及很多指针操作，对任何一个指针都进行检测并判断它们是否已经越出了内存映射文件的范围是很麻烦的，使用 SEH 可以让这方面的工作开销最少。

```

assume     esi:ptr IMAGE_DOS_HEADER
.if
    [esi].e_magic != IMAGE_DOS_SIGNATURE
    jmp     _ErrFormat
.endif
add        esi,[esi].e_lfanew
assume     esi:ptr IMAGE_NT_HEADERS
.if
    [esi].Signature != IMAGE_NT_SIGNATURE
    jmp     _ErrFormat
.endif
invoke     ProcessPeFile,@lpMemoryv.esi,@dwFileSize

```

ProcessPeFile.asm 文件的内容如下所示:

```

;*****
; 获取节的名称，由于节名称不一定是以 0 结尾的，所以要进行一些处理
;*****

```

程序首先显示了 PE 文件头中一些重要字段的数值，如 Machine、NumberOfSections 和 Characteristics 等字段，然后根据 NumberOfSections 的数值构造一个循环，并在循环中显示节的信息。

读者可以用这个 PEInfo 程序打开不同的文件来验证一下本节中叙述的一些内容，并思考下面的问题，由此可以印证很多本节中讲述的内容：

- 643

17.2 导 入 表

在开始下面几节的介绍前，先来复习一下 17.1 节中提出的两个概念。

首先，PE 文件中的数据按照装入内存后的页面属性被划分成多个节，并由节表中的数据来描述这些节。一个节中的数据仅仅是属性相同而已，并不一定就是同一种用途的，比如导入表、导出表等就有可能和只读常量一起被放在同一个节中，因为它们的属性同是可读不可写的。

其次，由于不同用途的数据可能被放在同一个节中，仅仅依靠节表是无法确定它们的存放位置的，PE 文件中依靠文件头中 IMAGE_OPTIONAL_HEADER32 结构内的数据目录表来指出它们的位置，可以由数据目录表来定位的数据包括导入表、导出表、资源、重定位表和 TLS 等 15 种数据。

好了，现在要引出这几节将要讲述的内容了：从数据目录表得到的仅仅是这些数据的 RVA 和数据块的大小，很明显，不同的数据块中的数据组织方式是不同的，比如导入表和资源数据块中的数据就完全是两码事情，要想深入了解 PE 文件就必须了解这些数据的组织方式，以及系统是如何处理它们的，这就是本节以及下面几节的内容。

本节将首先介绍导入表的格式，下面的几节将逐一介绍导出表、资源和重定位表的格式和使用方法。

17.2.1 导入表简介

在 Win32 编程中常常用到“导入函数”（Import functions），导入函数就是被程序调用但其执行代码又不在程序中的函数，这些函数的代码位于一个或者多个 DLL 中，在调用者程序中只保留一些函数信息，包括函数名及其驻留的 DLL 名等。

对于存储在磁盘上的 PE 文件来说，它无法得知这些导入函数会在内存的哪个地方出现，只有当 PE 文件被装入内存的时候，Windows 装载器才将 DLL 装入，并将调用导入函数的指令和函数实际所处的地址联系起来，这就是“动态链接”的概念。动态链接是通过 PE 文件中定义的“导入表”（Import Table）来完成的，导入表中保存的正是函数名和其驻留的 DLL 名等动态链接所必需的信息。

1. 调用导入函数的指令

程序被执行的时候是怎样使用导入函数的呢？先将第 3 章中那个简单的 Hello World 程序反汇编一把，看看调用导入函数的指令都是什么样子的，需要反汇编的两句源代码如下：

invoke	MessageBox, NULL, offset szText, offset szCaption, MB_OK
invoke	ExitProcess, NULL

当使用 W32Dasm 反汇编以后，这两句代码变成了以下的指令：

:00401000 6A00	push 00000000
:00401002 6800304000	push 00403000
:00401007 680F304000	push 0040300F
:0040100C 6A00	push 00000000

```

:0040100E E807000000      Call 0040101A      ;MessageBox
:00401013 6A00                push 00000000
:00401015 E806000000      Call 00401020      ;ExitProcess
:0040101A FF2508204000      jmp dword ptr [00402008]
:00401020 FF2500204000      jmp dword ptr [00402000]

```

反汇编后，对 MessageBox 和 ExitProcess 函数的调用变成了对 0040101A 和 00401020 地址的调用，但是这两个地址显然是位于程序自身模块而不是在 DLL 模块中的，实际上，这是由编译器在程序所有代码的后面自动加上的 `Jmp dword ptr [xxxxxxx]` 类型的指令，这个指令是一个间接寻址的跳转指令，xxxxxxx 地址中存放的才是真正的导入函数的地址。在这个例子中，00402000 地址处存放的就是 ExitProcess 函数的地址。

那么在没有装载到内存之前，PE 文件中的 00402000 地址处的内容是什么呢？使用在 17.1.4 节中了解的方法来查看一下。

首先，使用 17.1.4 节的例子文件 PEInfo.exe 去查看一下 Hello.exe 文件，会得到以下的信息：

文件名: C:\Documents and Settings\Administrator\桌面\Hello.exe

```

运行平台:      0x014C
节区数量:      3
文件标记:      0x010F
建议装入地址:  0x00400000

```

节区名称	节区大小	虚拟地址	Raw_尺寸	Raw_偏移	节区属性
.text	00000026	00001000	00000200	00000400	60000020
.rdata	00000092	00002000	00000200	00000600	40000040
.data	00000022	00003000	00000200	00000800	C0000040

由于建议装入地址是 00400000h，所以 00402000h 地址实际上处于 RVA 为 2000h 的地方，再看看各个节的虚拟地址，可以发现 2000h 开始的地方位于 .rdata 节内，而这个节的 Raw_偏移项目为 600h，也就是说 00402000h 地址的内容实际上对应 PE 文件中偏移 600h 处的数据。

现在随便找一个十六进制编辑器来看看文件 0600h 处的内容是什么：

```

0600 76 20 00 00 00 00 00 00-5C 20 00 00 00 00 00 00  v ..... \ .....
0610 54 20 00 00 00 00 00 00-00 00 00 00 6A 20 00 00  T ..... j ..
0620 08 20 00 00 00 4C 20 00 00-00 00 00 00 00 00 00  . ..L .....
0630 84 20 00 00 00 20 00 00-00 00 00 00 00 00 00 00  . ... .....
0640 00 00 00 00 00 00 00 00 00-00 00 00 00 76 20 00 00  ..... v ..
0650 00 00 00 00 5C 20 00 00-00 00 00 00 BB 01 4D 65  .... \ ..... Me
0660 73 73 61 67 65 42 6F 78-41 00 55 53 45 52 33 32  ssageBoxA.USER32
0670 2E 64 6C 6C 00 00 75 00-45 78 69 74 50 72 6F 63  .dll..u.ExitProc
0680 65 73 73 00 4B 45 52 4E-45 4C 33 32 2E 64 6C 6C  ess.KERNEL32.dll
0690 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....

```

查看的结果是 00002076h，这显然不会是内存中的 ExitProcess 函数的地址，慢着！将它作为 RVA 看会怎么样呢？查看节表可以发现 RVA 地址 00002076h 也处于 .rdata 节内，减去节的起始地址 00002000h 后得到这个 RVA 相对于节首的偏移是 76h，也就是说它对应文件 0676h 开

始的地方，接下来可以惊奇地发现，0676h 再过去两个字节的内容正是函数名字字符串“ExitProcess”！

这都有点搞糊涂了，Call ExitProcess 指令被编译成了 Call aaaaaaaa 类型的指令，而 aaaaaaaa 处的指令是 `Jmp dword ptr [xxxxxxx]`，而 xxxxxxxx 地址的地方只是一个似乎是指向函数名字字符串的 RVA 地址，这一系列的指令显然是无法正确执行的！

但如果告诉你，当 PE 文件被装载的时候，Windows 装载器会根据 xxxxxxxx 处的 RVA 得到函数名，再根据函数名在内存中找到函数地址，并且用函数地址将 xxxxxxxx 处的内容替换成真正的函数地址，那么所有的疑惑就迎刃而解了。

接下来看看如何去获取导入表的位置，以及导入表中的数据是如何组织以便 Windows 装载器能够顺利地进行上面的转换工作的。

2. 获取导入表的位置

导入表的位置和大小可以从 PE 文件头中 IMAGE_OPTIONAL_HEADER32 结构的数据目录字段中获取，对应的项目是 DataDirectory 字段的第 2 个 IMAGE_DATA_DIRECTORY 结构（如表 17.4 所示）。

从 IMAGE_DATA_DIRECTORY 结构的 VirtualAddress 字段得到的是导入表的 RVA 值，如果在内存中查找导入表，那么将 RVA 值加上 PE 文件装入的基址就是实际的地址；如果在 PE 文件中查找导入表，那么需要首先使用 17.1.4 节中例举的_RVAToOffset 子程序将 RVA 首先转换成文件偏移。

17.2.2 导入表的结构

1. PE 文件中的导入表

现在得到了包含导入表的数据块，导入表由一系列的 IMAGE_IMPORT_DESCRIPTOR 结构组成，结构的数量取决于程序要使用的 DLL 文件的数量，每个结构对应一个 DLL 文件，例如，如果一个 PE 文件从 10 个不同的 DLL 文件中引入了函数，那么就存在 10 个 IMAGE_IMPORT_DESCRIPTOR 结构来描述这些 DLL 文件，在所有这些结构的最后，由一个内容全为 0 的 IMAGE_IMPORT_DESCRIPTOR 结构作为结束。

IMAGE_IMPORT_DESCRIPTOR 结构的定义如下：

```

IMAGE_IMPORT_DESCRIPTOR STRUCT
    union
        Characteristics    dd    ?
        OriginalFirstThunk dd    ?
    ends
    TimeDateStamp          dd    ?
    ForwarderChain          dd    ?
    Name1                   dd    ?
    FirstThunk              dd    ?
IMAGE_IMPORT_DESCRIPTOR ENDS

```

结构中的 Name1 字段(使用 Name1 作为字段名同样是因为 Name 一词和 MASM 的关键字冲突)

是一个 RVA，它指向此结构所对应的 DLL 文件的名称，这个文件名是一个以 NULL 结尾的字符串。

OriginalFirstThunk 字段和 FirstThunk 字段的含义现在可以看成是相同的（使用“现在”一词的含义马上会见分晓），它们都指向一个包含一系列 IMAGE_THUNK_DATA 结构的数组，数组中的每个 IMAGE_THUNK_DATA 结构定义了一个导入函数的信息，数组的最后以一个内容为 0 的 IMAGE_THUNK_DATA 结构作为结束。

一个 IMAGE_THUNK_DATA 结构实际上就是一个双字，之所以把它定义成结构，是因为它在不同的时刻有不同的含义，结构的定义如下：

```

IMAGE_THUNK_DATA STRUCT
    union ul
        ForwarderString dd    ?
        Function dd          ?
        Ordinal dd           ?
        AddressOfData dd     ?
    ends
IMAGE_THUNK_DATA ENDS

```

一个 IMAGE_THUNK_DATA 结构如何用来指定一个导入函数呢？当双字（就是指结构！）的最高位为 1 时，表示函数是以序号的方式导入的，这时双字的低位就是函数的序号。读者可以用预定义值 IMAGE_ORDINAL_FLAG32（或 80000000h）来对最高位进行测试，当双字的最高位为 0 时，表示函数以字符串类型的函数名方式导入，这时双字的值是一个 RVA，指向一个用来定义导入函数名称的 IMAGE_IMPORT_BY_NAME 结构，此结构的定义如下：

```

IMAGE_IMPORT_BY_NAME STRUCT
    Hint      dw    ?
    Name1 db    ?
IMAGE_IMPORT_BY_NAME ENDS

```

结构中的 Hint 字段也表示函数的序号，不过这个字段是可选的，有些编译器总是将它设置为 0，Name1 字段定义了导入函数的名称字符串，这是一个以 0 为结尾的字符串。

整个过程听起来有些复杂，其实以图 17.5 来说明就很清楚了，图中示意了可执行文件导入了 Kernel32.dll 中的 ExitProcess、ReadFile、WriteFile 和一个以序号为导入方式的函数（姑且称这个函数为 ImportByNo）的情况，假设这 4 个函数的序号分别是 02f6h、0111h、002bh 和 0010h。

现在来分析一下图 17.5 中的示例，导入表中 IMAGE_IMPORT_DESCRIPTOR 结构的 Name1 字段指向字符串“Kernel32.dll”，表明当前要从 Kernel32.dll 文件中导入函数，OriginalFirstThunk 和 FirstThunk 字段指向两个同样的 IMAGE_THUNK_DATA 数组，由于要导入的是 4 个函数，所以数组中包含 4 个有效项目并以最后一个内容为 0 的项目作为结束。

第 4 个函数是以序号导入的，与其对应的 IMAGE_THUNK_DATA 结构的最高位等于 1，和函数的序号 0010h 组合起来的数值就是 80000010h，其余的 3 个函数采用的是以函数名导入的方式，所以 IMAGE_THUNK_DATA 结构的数值是一个 RVA，分别指向 3 个 IMAGE_IMPORT_BY_NAME 结构，每个 IMAGE_IMPORT_BY_NAME 结构的第一个字段是函数的序号，后面就是函数的字符串名称了，一切就是这么简单！

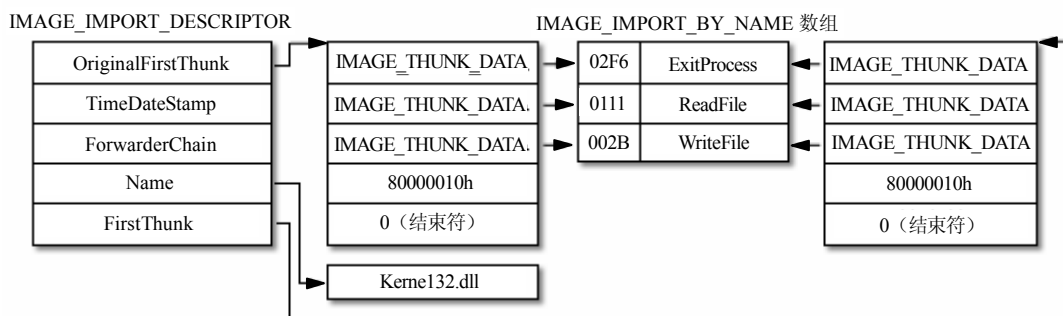


图 17.5 函数的导入方法举例

2. 内存中的导入表

为什么需要两个一模一样的 `IMAGE_THUNK_DATA` 数组呢？答案是当 PE 文件被装入内存的时候，其中一个数组的值将被改作他用，还记得前面分析 Hello World 程序时提到的吗？Windows 装载器会将指令 `Jmp dword ptr [xxxxxxx]` 指定的 xxxxxxxx 处的 RVA 替换成真正的函数地址，其实 xxxxxxxx 地址正是由 `FirstThunk` 字段指向的那个数组中的一员。

实际上，当 PE 文件被装入内存后，内存中的映像就被 Windows 装载器修正成了图 17.6 所示的样子，其中由 `FirstThunk` 字段指向的那个数组中的每个双字都被替换成了真正的函数入口地址，之所以在 PE 文件中使用两份 `IMAGE_THUNK_DATA` 数组的拷贝并修改其中的一份，是为了最后还可以留下一份拷贝用来反过来查询地址所对应的导入函数名。

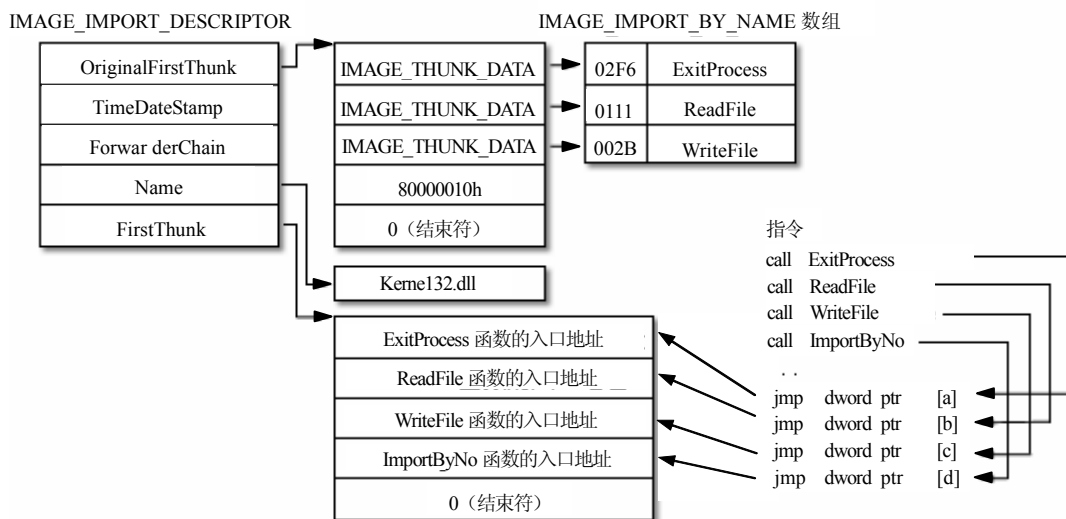




图 17.6 导入表被装入内存后的样子


3. 导入地址表 (IAT)

`IMAGE_IMPORT_DESCRIPTOR` 结构中 `FirstThunk` 字段指向的数组最后会被替换成导入函数的真正入口地址，暂且把这个数组称为导入地址数组。在 PE 文件中，所有 DLL 对应的导入地址数组在位置上是被排列在一起的，全部这些数组的组合也被称为导入地址表 (Import Address



器

器



器

```

        mov     edi, eax
        assume  edi:ptr IMAGE_IMPORT_DESCRIPTOR
;*****
; 显示 PE 文件名
;*****
        invoke  _GetRVASection, _lpFile, [edi].OriginalFirstThunk
        invoke  wsprintf, addr @szBuffer, addr szMsg, addr szFileName, eax
        invoke  SetWindowText, hWinEdit, addr @szBuffer
;*****
; 循环处理 IMAGE_IMPORT_DESCRIPTOR 直到遇到全零的则结束
;*****
.while  [edi].OriginalFirstThunk || [edi].TimeDateStamp || \
        [edi].ForwarderChain || [edi].Name1 || [edi].FirstThunk
        invoke  _RVAToOffset, _lpFile, [edi].Name1
        add     eax, _lpFile
        invoke  wsprintf, addr @szBuffer, addr szMsgImport, eax, \
        [edi].OriginalFirstThunk, [edi].TimeDateStamp, \
        [edi].ForwarderChain, [edi].FirstThunk
        invoke  _AppendInfo, addr @szBuffer
;*****
; 获取 IMAGE_THUNK_DATA 列表地址 --> ebx
;*****
        .if     [edi].OriginalFirstThunk
            mov     eax, [edi].OriginalFirstThunk
        .else
            mov     eax, [edi].FirstThunk
        .endif
        invoke  _RVAToOffset, _lpFile, eax
        add     eax, _lpFile
        mov     ebx, eax
;*****
; 循环处理所有的 IMAGE_THUNK_DATA
;*****
        .while  dword ptr [ebx]
;*****
; 按序号导入
;*****
        .if     dword ptr [ebx] & IMAGE_ORDINAL_FLAG32
            mov     eax, dword ptr [ebx]
            and     eax, 0FFFFh
            invoke  wsprintf, addr @szBuffer, \
            addr szMsgOrdinal, eax
        .else
;*****
; 按函数名导入
;*****
            invoke  _RVAToOffset, _lpFile, dword ptr [ebx]
            add     eax, _lpFile
            assume  eax:ptr IMAGE_IMPORT_BY_NAME
            movzx   ecx, [eax].Hint
            invoke  wsprintf, addr @szBuffer, \
            addr szMsgName, ecx, addr [eax].Name1

```

```

                assume    eax:nothing
            .endif
            invoke    _AppendInfo, addr @szBuffer
            add        ebx, 4
        .endw
        add        edi, sizeof IMAGE_IMPORT_DESCRIPTOR
    .endw
;*****
_Ret:
                assume    edi:nothing
                popad
                ret

_ProcessPeFile    endp
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

```

如果读者已经理解了 17.2.1 和 17.2.2 节中的内容，那么这段源程序是很容易看懂的，当 `_ProcessPeFile` 子程序在 `Main.asm` 中被调用的时候，输入参数 `lpFile` 和 `lpPeHead` 已经预先指向了被读到内存中的可执行文件头部和 PE 文件头的位置。程序首先查找数据目录表并得到导入表的地址，然后循环处理导入表中的每个 `IMAGE_IMPORT_DESCRIPTOR` 结构。

对于每个 `IMAGE_IMPORT_DESCRIPTOR` 结构，程序首先显示 `Name1` 字段指向的 DLL 文件名；然后继续构造一个循环来处理 `OriginalFirstThunk` 指向的 `IMAGE_THUNK_DATA` 数组，程序使用预定义值 `IMAGE_ORDINAL_FLAG32` 来测试 `IMAGE_THUNK_DATA` 结构的最高位，当最高位为 1 时显示导入函数的序号，当最高位为 0 时则显示导入函数的 Hint 值和函数名称。

在这个程序中读者经常提及的一个问题是为什么访问数据目录表的时候地址是 `DataDirectory[8].VirtualAddress`，而不是 `DataDirectory[1].VirtualAddress`，要知道，数据目录表中导入表的索引是 1 呀？答案是：这是 MASM 的语法决定的，在 MASM 中，不管数组中的单个数组项字节数是多少，括号中的数值都是字节地址而不是数组下标，所以数据目录表结构的长度是 8 的时候，访问第 `n` 个结构时要寻址的就是 `DataDirectory[n*8]`，所以上面的 `DataDirectory[8]` 实际上是 `DataDirectory[1*8]` 的意思，表示访问的是索引号为 1 的数组项。

17.3 导出表

当 PE 文件被执行的时候，Windows 装载器将文件装入内存并将导入表中登记的 DLL 文件一并装入，再根据 DLL 文件中的函数导出信息对被执行文件的 IAT 表进行修正。在这些包含导出函数的 DLL 文件中，导出信息被保存在导出表中，通过导出表，DLL 文件向系统提供导出函数的名称、序号和入口地址等信息，以便 Windows 装载器通过这些信息来完成动态链接的过程。

扩展名为 `.exe` 的 PE 文件中一般不存在导出表，而大部分的 `.dll` 文件中都包含导出表，但是这并不是必然的，比如，用做纯资源的 `.dll` 文件就不提供导出函数，文件中也就不存在导出表；另外，偶尔也可以见到包含导出函数和导出表的 `.exe` 文件。本节将介绍导出表的结构和使用方法。

17.3.1 导出表的结构

1. 获取导出表的位置

导出表的位置和大小同样可以从 PE 文件头中的数据目录中获取，与导出表对应的项目是数据目录中的首个 IMAGE_DATA_DIRECTORY 结构，从这个结构的 VirtualAddress 字段得到的就是导出表的 RVA 值。如果在磁盘上的 PE 文件中查找导出表，那么使用_RVAToOffset 子程序将 RVA 转换成文件偏移就可以了。

2. 导出表的组成

导出表的功能是与导入表配合使用的，既然在导入表中可以用函数名或序号来导入，那么可以想象，导出表中必然也可以用函数名或序号这两种方法来导出函数。事实的确如此，导出表中为每个导出函数定义了导出序号，但函数名的定义是可选的。对于定义了函数名的函数来说，既可以使用名称导出，也可以使用序号导出；对于没有定义函数名的函数来说，只能使用序号来导出。

导出表的起始位置有一个 IMAGE_EXPORT_DIRECTORY 结构，与导入表中有多个 IMAGE_IMPORT_DESCRIPTOR 结构不同，导出表中只有一个 IMAGE_EXPORT_DIRECTORY 结构，这个结构的定义如下：

IMAGE_EXPORT_DIRECTORY STRUCT			
Characteristics	DWORD	?	;未使用，总是为 0
TimeDateStamp	DWORD	?	;文件的产生时刻
MajorVersion	WORD	?	;未使用，总是为 0
MinorVersion	WORD	?	;未使用，总是为 0
nName	DWORD	?	;指向文件名的 RVA
nBase	DWORD	?	;导出函数的起始序号
NumberOfFunctions	DWORD	?	;导出函数的总数
NumberOfNames	DWORD	?	;以名称导出的函数总数
AddressOfFunctions	DWORD	?	;指向导出函数地址表的 RVA
AddressOfNames	DWORD	?	;指向函数名地址表的 RVA
AddressOfNameOrdinals	DWORD	?	;指向函数名序号表的 RVA
IMAGE_EXPORT_DIRECTORY ENDS			

这个结构中的一些字段并没有被使用，其余有意义的字段说明如下，读者可以参考图 17.7 来理解这些字段之间的关系。

● nName 字段

这个字段是一个 RVA 值，指向一个定义了模块名称的字符串。这个字符串说明了模块的原始文件名，比如说，即使 Kernel32.dll 文件被改名为 Ker.dll，仍然可以从这个字符串中的值得知它被编译时的文件名是“Kernel32.dll”。

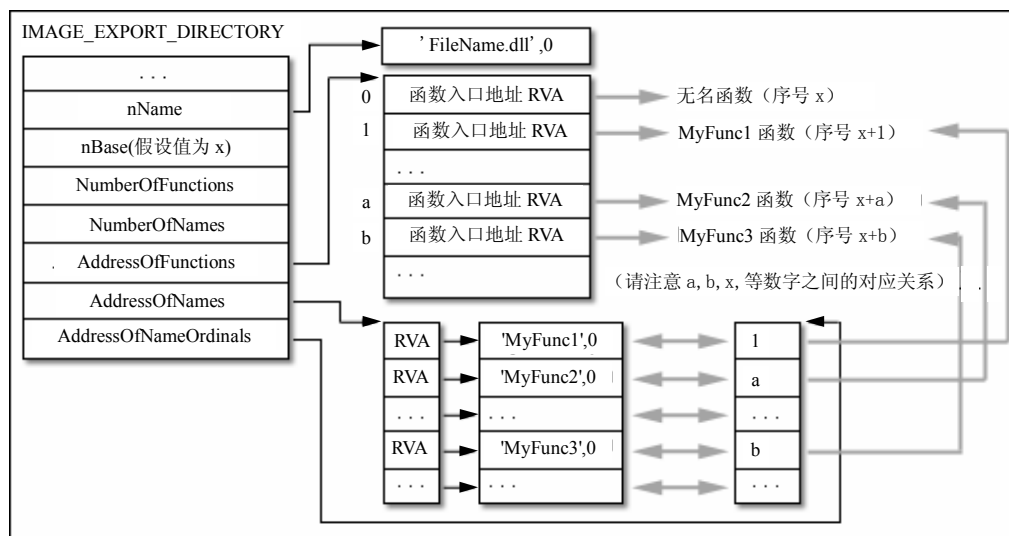


图 17.7 函数导出的示意图

- `NumberOfFunctions` 字段

文件中包含的导出函数的总数。

- `NumberOfNames` 字段

被定义了函数名称的导出函数的总数。显然，只有这个数量的函数既可以用函数名方式导出，也可以用序号方式导出，剩下的 `NumberOfFunctions` 减去 `NumberOfNames` 数量的函数只能用序号方式导出。`NumberOfNames` 字段的值只会小于或者等于 `NumberOfFunctions` 字段的值，如果这个值是 0，表示所有的函数都是以序号方式导出的。

- `AddressOfFunctions` 字段

这是一个 RVA 值，指向包含全部导出函数入口地址的双数组，数组中的每一项是一个 RVA 值，数组的项数等于 `NumberOfFunctions` 字段的值。

- `nBase` 字段

导出函数序号的起始值。将 `AddressOfFunctions` 字段指向的入口地址表的索引号加上这个起始值就是对应函数的导出序号，举例来说，假如 `nBase` 字段的值为 x ，那么入口地址表指定的第一个导出函数的序号就是 x ，第二个导出函数的序号就是 $x+1$ ，总之，一个导出函数的导出序号等于 `nBase` 字段的值加上其在入口地址表中的位置索引值。

- `AddressOfNames` 和 `AddressOfNameOrdinals` 字段

`AddressOfNames` 字段的数值是一个 RVA 值，指向函数名字符串地址表，这个地址表是一个双数组，数组中的每一项指向一个函数名称字符串的 RVA，数组的项数等于 `NumberOfNames` 字段的值，所有有名称的导出函数的名称字符串都定义在这个表中。

那么这些函数名称究竟对应地址表中的那个函数呢？`AddressOfNameOrdinals` 字段就派上用场了，这个字段也是一个 RVA 值，指向另一个 word 类型的数组（注意不是双数组），数

组的项目与文件名地址表中的项目一一对应，项目的值代表函数入口地址表的索引，这样函数名称与函数入口地址就关联起来了。

举例说明，假如函数名称字符串地址表的第 n 项指向一个字符串“MyFunction”，那么可以去查找 AddressOfNameOrdinals 字段指向的数组的第 n 项，假如第 n 项中存放的值是 x ，表示 AddressOfFunctions 字段描述的地址表中的第 x 项函数入口地址（假定入口地址值是 aaaa）对应的函数名就是“MyFunction”，这时这个函数的全部信息就可以如下描述。

函数名称：MyFunction，导出序号：nBase 的值 + x ，入口地址：aaaa

可以看到，AddressOfNameOrdinals 字段描述的数组仅仅起了一个桥梁的作用。

3. 从序号查找入口地址

下面来模拟一下 Windows 装载器查找导出函数入口地址的过程。如果已知函数的导出序号，如何得到入口地址呢？

步骤如下所示：

（1）定位到 PE 文件头。

（2）从 PE 文件头中的 IMAGE_OPTIONAL_HEADER32 结构中取出数据目录表，并从第一个数据目录中得到导出表的地址。

（3）从导出表的 nBase 字段得到起始序号。

（4）将需要查找的导出序号减去起始序号，得到函数在入口地址表中的索引。

（5）检测索引值是否大于导出表的 NumberOfFunctions 字段的值，如果大于后者的话，说明输入的序号是无效的。

（6）用这个索引值在 AddressOfFunctions 字段指向的导出函数入口地址表中取出相应的项目，这就是函数的入口地址 RVA 值，当函数被装入内存的时候，这个 RVA 值加上模块实际装入的基址，就得到了函数真正的入口地址。

4. 从函数名称查找入口地址

如果已知函数的名称，如何得到函数的入口地址呢？与使用序号来获取入口地址相比，这个过程要相对复杂一点：

（1）最初的步骤是一样的，那就是首先得到导出表的地址。

（2）从导出表的 NumberOfNames 字段得到已命名函数的总数，并以这个数字作为循环的次数来构造一个循环。

（3）从 AddressOfNames 字段指向的函数名称地址表的第一项开始，在循环中将每一项定义的函数名与要查找的函数名相比较，如果没有任何一个函数名是符合的，表示文件中没有指定名称的函数。

（4）如果某一项定义的函数名与要查找的函数名符合，那么记下这个函数名在字符串地址表中的索引值，然后在 AddressOfNameOrdinals 指向的数组中以同样的索引值取出数组项的

哭

```

        .if      ! eax
        invoke   MessageBox, hWinMain, \
                addr szErrNoExport, NULL, MB_OK
        jmp      _Ret
    .endif
    invoke   _RVAToOffset, _lpFile, eax
    add      eax, _lpFile
    mov      edi, eax
;*****
; 显示一些常用的信息
;*****
        assume   edi:ptr IMAGE_EXPORT_DIRECTORY
        invoke   _RVAToOffset, _lpFile, [edi].nName
        add      eax, _lpFile
        mov      ecx, eax
        invoke   _GetRVASection, _lpFile, [edi].nName
        invoke   wsprintf, addr @szBuffer, addr szMsg, \
                addr szFileName, eax, ecx, [edi].nBase, \
                [edi].NumberOfFunctions, [edi].NumberOfNames, \
                [edi].AddressOfFunctions, [edi].AddressOfNames, \
                [edi].AddressOfNameOrdinals
        invoke   SetWindowText, hWinEdit, addr @szBuffer
;*****
        invoke   _RVAToOffset, _lpFile, [edi].AddressOfNames
        add      eax, _lpFile
        mov      @lpAddressOfNames, eax
        invoke   _RVAToOffset, _lpFile, [edi].AddressOfNameOrdinals
        add      eax, _lpFile
        mov      @lpAddressOfNameOrdinals, eax
        invoke   _RVAToOffset, _lpFile, [edi].AddressOfFunctions
        add      eax, _lpFile
        mov      esi, eax          ;esi --> 函数地址表
;*****
; 循环显示导出函数的信息
;*****
        mov      ecx, [edi].NumberOfFunctions
        mov      @dwIndex, 0
        @@:
        pushad
;*****
; 在按名称导出的索引表中
;*****
        mov      eax, @dwIndex
        push     edi
        mov      ecx, [edi].NumberOfNames
        cld
        mov      edi, @lpAddressOfNameOrdinals
        repnz    scasw
        .if     ZERO?      ;找到函数名称
            sub     edi, @lpAddressOfNameOrdinals
            sub     edi, 2
            shl     edi, 1
            add     edi, @lpAddressOfNames
            invoke   _RVAToOffset, _lpFile, dword ptr [edi]
        .endif

```


程序一开始首先从数据目录表获取导出表的地址，然后显示一些文件的信息，如导出表中定义的原始文件名和导出表中的一些字段的值。

按照 NumberOfFunctions 的数量循环处理导入函数地址表中的每个项目，在循环中每次用当前项目的索引值在 AddressOfNameOrdinals 指定的数组中查找，如果索引值没有包括在数组中，表示没有任何已定义的函数名称符合当前的函数入口地址，这时程序将显示“按照序号导出”的信息；如果在数组中找到当前索引值，表明有个函数名称对应当前的函数入口地址，程序记下在数组中找到当前索引的位置，并在 AddressOfNames 字段定义的函数名地址表的同样位置取出函数名称字符串的 RVA，最后以这个 RVA 得到函数名称字符串并显示出来。

```
mov     edi, @lpAddressOfNameOrdinals
repnz   scasw
.if     ZERO?      ; 找到函数名称
sub     edi, @lpAddressOfNameOrdinals
sub     edi, 2
shl     edi, 1
add     edi, @lpAddressOfNames
```

657

到的项目后面一个 word 的位置，将 edi 减去数组的基址并减去 2（一个 word 的长度），得到的就是找到的项目的位置偏移。由于这个数组是 word 类型的，而 AddressOfNames 指向的数组是 dword 类型的，所以还要将偏移乘以 2 来修正一下（使用 shl edi, 1 指令），用修正后的偏移在 AddressOfNames 表中就可以得到指向函数名称字符串的 RVA 了。

17.4 资源

17.4.1 资源简介

资源是 PE 文件中非常重要的部分，几乎所有的 PE 文件中都包含资源，与导入表和导出表相比，资源的组织方式要复杂得多，读者只要看看图 17.8 中的示意图，就知道笔者所言不虚。

如果一开始就扎进一堆与资源相关的数据结构中去分析各字段的含义，恐怕会越来越糊涂，要了解资源的话，重在理解资源整体上的组织结构。

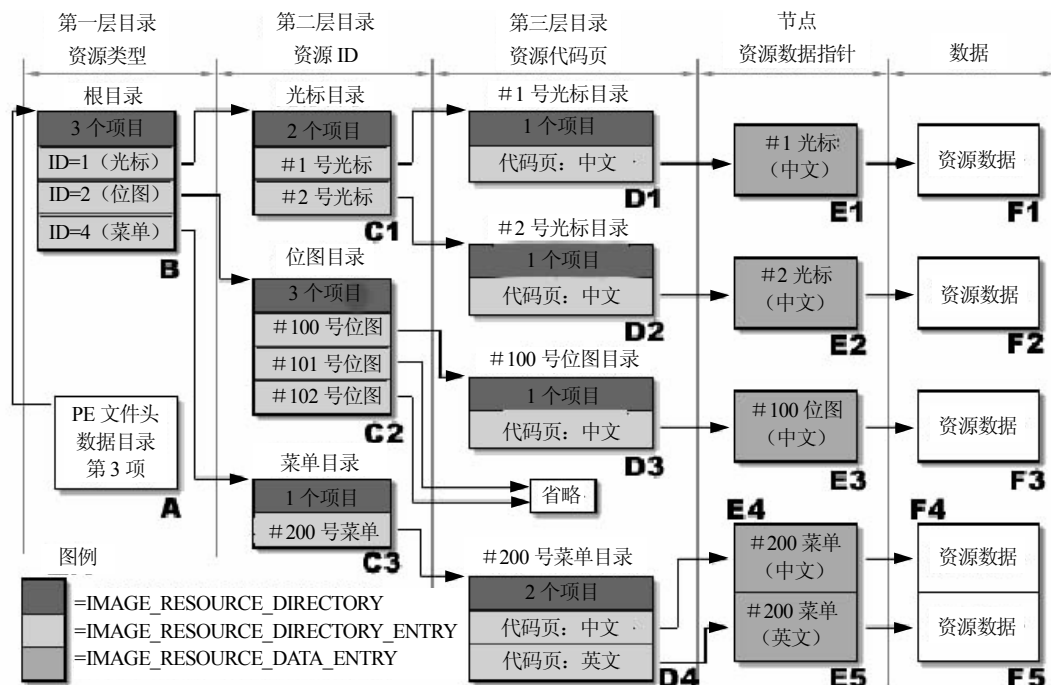


图 17.8 PE 文件中资源的组织方式

我们知道，PE 文件资源中的内容包括光标、图标、位图、菜单等十几种标准的类型，除此之外，还可以使用自定义的类型（这些类型的资源在第 5 章中已经有所介绍）。每种类型的资源中可能存在多个资源项，这些资源项用不同的 ID 或者名称来分辨，在某个资源 ID 下，还可以同时存在不同代码页的版本。

要将这么多类型的不同 ID 的资源有序地组织起来，用类似于磁盘目录结构的方式是很不错的。打个比方，假如在磁盘的根目录下按照类型建立若干个第 2 层目录，目录名是“光标”、

“图标”、“位图”和“菜单”等，就可以将各种资源分类放入这些目录中，假设现在有 n 个光标，那么在“光标”目录中再以光标 ID 为名建立 n 个第 3 层子目录，进入某个子目录后，再以代码页为名称建立不同文件，这样所有的资源就按照树型目录的方式组织起来了。现在要查找某个资源的话，那么按照根目录→资源类型→资源 ID→资源代码页这样的步骤一层层地进入相应的子目录并找到正确的资源。

如图 17.8 所示，PE 文件中组织资源的方式与上面的构思极其相似，正是按照根目录→资源类型→资源 ID 的 3 层树型目录结构来组织资源的，只不过在第 3 层目录中放置的代码页“文件”不是资源本身而是一个用来描述资源的结构罢了，通过这个结构中的指针才能最后找到资源数据。

17.4.2 资源的组织方式

1. 获取资源的位置

图 17.8 所示的所有目录结构数据和所有的资源数据都集中放在一个资源数据块中，资源数据块的位置和大小可以从 PE 文件头中的 IMAGE_OPTIONAL_HEADER32 结构的数据目录字段中获取，与资源对应的项目是数据目录中的第 3 个 IMAGE_DATA_DIRECTORY 结构（如表 17.4 所示），从这个结构的 VirtualAddress 字段得到的就是资源块地址的 RVA 值。

如图 17.8 中的 A 所示，从数据目录表中得到的资源块的起始地址就是资源根目录的起始地址，从这里开始就可以一层层地找到资源的所有信息了。

在获取资源块地址的时候，注意不要使用查找“.rsrc”节起始地址的方法，虽然在一般情况下资源总是在“.rsrc”节中，但这并不是必然的。

2. 资源目录

好了，现在继续深入一步，资源目录树的根目录地址已经得到了，那么整个目录树上的目录是如何描述的呢？注意图 17.8 左下角的图例在整个目录树中出现的位置，这样就可以发现：不管是根目录，还是第 2 层或第 3 层中的每个目录都是由一个 IMAGE_RESOURCE_DIRECTORY 结构和紧跟其后的数个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构组成的，这两种结构一起组成了一个目录块。

IMAGE_RESOURCE_DIRECTORY 结构中包含的是本目录的各种属性信息，其中有两个字段说明了本目录中的目录项数量，也就是后面的 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构的数量。

IMAGE_RESOURCE_DIRECTORY 结构的定义如下所示：

IMAGE_RESOURCE_DIRECTORY STRUCT			
Characteristics	dd	?	;理论上为资源的属性，不过事实上总是 0
TimeDateStamp	dd	?	;资源的产生时刻
MajorVersion	dw	?	;理论上为资源的版本，不过事实上总是 0
MinorVersion	dw	?	
NumberOfNamedEntries	dw	?	;以名称命名的入口数量
NumberOfIdEntries	dw	?	;以 ID 命名的入口数量
IMAGE_RESOURCE_DIRECTORY ENDS			

在这个结构中，最重要的是最后两个字段，它们说明了本目录中目录项的数量，那么为什么有两个字段呢？

原因是这样的：不管是资源种类，还是资源名称都可以用名称或者 ID 两种方式定义，比如，在*.rc 文件中这样定义：

100	ICON	"Test.ico"	// (例 1)
101	WAVE	"Test.wav"	// (例 2)
HelpFile	HELP	"Test.chm"	// (例 3)
102	12345	"Test.bin"	// (例 4)

例 1 定义了一个资源 ID 为 100 的光标资源，其资源类型为“ICON”，但“ICON”只是一个用在 rc 文件中的助记符，在资源编译器里面会被换成数值型的类型 ID，所有的标准类型都是以数值型 ID 定义的，在资源定义中，1 到 10h 的 ID 编号保留给标准类型使用。

在例 2 中，标准的资源类型中并没有“WAVE”这一类型，这时资源的类型属于自定义型，类型的名称就是“WAVE”。

例 3 则定义了资源名称是“HelpFile”，类型名称为自定义字符串“HELP”的资源。

在例 4 中，资源的 ID 编号是 102，而类型则是数值型 ID，由于标准类型中并没有编号为 12345 的资源，所以这这也是一个自定义类型的资源。

在 IMAGE_RESOURCE_DIRECTORY 结构中，对以 ID 命名和以字符串命名的情况是分别指定的：NumberOfNamedEntries 字段是以字符串命名的资源数量，而 NumberOfIdEntries 字段的值是以 ID 命名的资源数量，所以两者的数量加起来才是本目录中的目录项总和，也就是当前 IMAGE_RESOURCE_DIRECTORY 结构后面紧跟的 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构的数量。

现在来介绍一下 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构，每个结构描述了一个目录项，IMAGE_RESOURCE_DIRECTORY_ENTRY 结构是这样定义的：

IMAGE_RESOURCE_DIRECTORY_ENTRY STRUCT	
Name1 dd ?	；目录项的名称字符串指针或 ID
OffsetToData dd ?	；目录项指针
IMAGE_RESOURCE_DIRECTORY_ENTRY ENDS	

结构中的两个字段说明如下：

● Name1 字段

这个字段的名称应该是“Name”，同样是因为和关键字冲突的原因改为“Name1”，它定义了目录项的名称或者 ID，这个字段的含义要看目录项用在什么地方，当结构用于第 1 层目录的时候（如图 17.8 中的 B 所示），这个字段定义的是资源的类型，也就是前面例子中的“ICON”，“WAVE”，“HELP”和 12345 等；当结构用于第 2 层目录的时候（如图 17.8 中的 C1 到 C3），这个字段定义的是资源的名称，也就是前面例子中的 100，101，“HelpFile”和 102 等；而当结构用于第 3 层目录的时候（如图 17.8 中的 D1 到 D4），这里定义的是代码页编号。

读者肯定会发现一个问题：当字段作为 ID 使用的时候，是可以放入一个双字的，如果使用字符串定义的时候，一个双字是不够的，这就需要将两种情况分别对待，区分的方法是使用

字段的最高位（位 31）。当位 31 是 0 的时候，表示字段的值作为 ID 使用；而位 31 为 1 的时候，字段的低位作为指针使用，但由于资源名称字符串是使用 UNICODE 来编码的，所以这个指针并不直接指向字符串，而是指向一个 IMAGE_RESOURCE_DIR_STRING_U 结构，这个结构包含 UNICODE 字符串的长度和字符串本身，其定义如下：

```

IMAGE_RESOURCE_DIR_STRING_U STRUCT
    Length1 dw    ?    ;字符串的长度
    NameString dw  ?    ;UNICODE 字符串, 由于字符串是不定长的, 所以这里只能
                        ;用一个 dw 表示, 实际上当长度为 100 的时候, 这里的数据
                        ;是 NameString dw 100 dup (?)
IMAGE_RESOURCE_DIR_STRING_U ENDS

```

如果要得到 ANSI 类型的以 0 结尾的字符串，需要将 NameString 字段中包括的 UNICODE 字符串用 WideCharToMultiByte 函数转换一下，具体的方法读者可以参考后面的例子。

● OffsetToData 字段

这个字段是一个指针，当它的最高位（位 31）为 1 时，低位数据指向下一层目录块的起始地址，也就是一个 IMAGE_RESOURCE_DIRECTORY 结构，这种情况一般出现在第 1 层和第 2 层目录中；当字段的位 31 为 0 时，指针指向的是用来描述资源数据块情况的 IMAGE_RESOURCE_DATA_ENTRY 指针，这种情况出现在第 3 层目录中。

当将 Name1 字段和 OffsetToData 用做指针时需要注意两点，首先是不要忘记将最高位清除（使用 7fffffffh 来 and 一下）；其次就是这两个指针是从资源块开始的地方算起的偏移量，也就是根目录的起始位置算起的偏移量。



注意：千万不要将这两个指针作为 RVA 来对待，否则会得到错误的地址。正确的计算方法是 将指针的值加上资源块首地址，结果才是真正的地址。

最后还需要说明的是，当 IMAGE_RESOURCE_DIRECTORY_ENTRY 用在第 1 层目录中的时候，它的 Name1 字段是作为资源类型来使用的。当资源类型以 ID 定义（最高位等于 0），并且 ID 数值在 1 到 16 之间时，表示这是系统预定义的类型，ID 与类型的对应关系请参考表 17.6；如果资源类型是以 ID 定义的并且数值在 16 以上，表示这是一个自定义的类型。

表 17.6 预定义的资源类型

类型 ID 值	在 Windows.inc 中的预定义值	资源类型
1	RT_CURSOR	光标(cursor)
2	RT_BITMAP	位图(bitmap)
3	RT_ICON	图标(icon)
4	RT_MENU	菜单(menu)
5	RT_DIALOG	对话框(dialog)
6	RT_STRING	字符串(string)
7	RT_FONTDIR	字体目录(font directory)

8	RT_FONT	字体(font)
9	RT_ACCELERATOR	加速键(accelerators)
10	RT_RCDATA	未格式化资源(unformatted)
11	(无)	消息表(message table)
12	RT_GROUP_CURSOR	光标组(group cursor)
14	RT_GROUP_ICON	图标组(group icon)
16	RT_VERSION	版本信息(version information)

3. 资源数据入口

沿着资源目录树按照根目录→资源类型→资源 ID 的顺序到达第 3 层目录后，这一层目录的 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构的 OffsetToData 字段指向的是一个 IMAGE_RESOURCE_DATA_ENTRY 结构（如图 17.8 中的 E1 到 E5 所示）。

IMAGE_RESOURCE_DATA_ENTRY 结构的定义如下所示：

IMAGE_RESOURCE_DATA_ENTRY STRUCT			
OffsetToData	dd?		;资源数据的 RVA
Size1	dd	?	;资源数据的长度
CodePage	dd	?	;代码页
Reserved	dd	?	;保留字段
IMAGE_RESOURCE_DATA_ENTRY ENDS			

IMAGE_RESOURCE_DATA_ENTRY 结构描述了资源数据所处的位置和大小，换句话说，就是经过了这么多层结构的长途跋涉以后，终于得到了某一个资源的详细信息。

结构中的 OffsetToData 字段的值是指向资源数据的指针，奇怪的是，这个指针却是一个 RVA 值，而不是以资源块的起始地址为基址的，这是读者需要特别注意的地方。Size1 字段的值是资源数据的大小。结构中的第 3 个字段是 CodePage，这个字段的名称有些奇怪，因为当前资源的代码页已经在第 3 层目录中指明了，在这里再定义一次有重复之嫌，在实际的应用中，这个字段好像未被使用，因为随便找一个 PE 文件看看就会发现这里的值总是为 0。

17.4.3 查看 PE 文件中的资源列表举例

本节中的例子遍历 PE 文件中的资源目录树并显示每个资源的详细信息，例子的源代码放在本书所附光盘的 Chapter17\Resource 目录中，同样，为了节省篇幅，界面代码沿用前面的 Main.asm 和 Main.rc 文件，下面是 Main.asm 中包括的 _ProcessPeFile.asm 文件的内容：

.const			
szMsg	db	' 文件名: %s', 0dh, 0ah	
	db	' -----', 0dh, 0ah	
	db	' 资源所处的节: %s', 0dh, 0ah, 0	
szErrNoRes	db	' 这个文件中没有包含资源!', 0	
szLevel1	db	0dh, 0ah	
	db	' -----', 0dh, 0ah	
	db	' 资源类型: %s', 0dh, 0ah	
	db	' -----', 0dh, 0ah, 0	
szLevel1byID	db	' %d (自定义编号)', 0	
szLevel2byID	db	' ID: %d', 0dh, 0ah, 0	

663

```

                                and     eax, 7fffffffh
                                add     eax, _lpRes
;*****
; 处理 IMAGE_RESOURCE_DIR_STRING_U 结构并将 UNICODE 转换成 ANSI 字符串
;*****
                                movzx   ecx, word ptr [eax]
                                add     eax, 2
                                mov     edx, eax
                                invoke   WideCharToMultiByte, CP_ACP, \
                                        WC_COMPOSITECHECK, edx, ecx, \
                                        addr @szResName, sizeof @szResName, \
                                        NULL, NULL
                                lea     eax, @szResName
                                .else
;*****
; 当资源类型为标准类型的时候查表得到资源的字符串说明
;*****
                                .if     eax <= 10h
                                    dec     eax
                                    mov     ecx, sizeof szType
                                    mul     ecx
                                    add     eax, offset szType
                                .else
                                    invoke  wsprintf, addr @szResName, \
                                            addr szLevel1byID, eax
                                    lea     eax, @szResName
                                .endif
                                .endif
                                invoke   wsprintf, addr @szBuffer, addr szLevel1, eax
;*****
; 第二层：资源 ID（或名称）
;*****
                                .elseif  _dwLevel == 2
                                    mov     edx, [esi].Name1
                                    .if     edx & 80000000h
;*****
; 资源以字符串方式命名
;*****
                                    and     edx, 7fffffffh
                                    add     edx, _lpRes
                                    movzx   ecx, word ptr [edx]
                                    add     edx, 2
                                    invoke   WideCharToMultiByte, CP_ACP, \
                                            WC_COMPOSITECHECK, edx, ecx, \
                                            addr @szResName, sizeof @szResName, \
                                            NULL, NULL
                                    invoke   wsprintf, addr @szBuffer, \
                                            addr szLevel2byName, addr @szResName
                                    .else
;*****
; 资源以 ID 命名
;*****
                                    invoke   wsprintf, addr @szBuffer, \
                                            addr szLevel2byID, edx

```



```

                                .endif
                .else
                                .break
                .endif
                invoke    _AppendInfo, addr @szBuffer
                invoke    _ProcessRes, _lpFile, _lpRes, ebx, @dwNextLevel
;*****
; 不是资源目录则显示资源详细信息（在第3层时的情况）
;*****
                .else
                        add     ebx, _lpRes
                        mov     ecx, [esi].Name1           ;代码页
                        assume   ebx:ptr IMAGE_RESOURCE_DATA_ENTRY
                        mov     eax, [ebx].OffsetToData
                        invoke    _RVAToOffset, _lpFile, eax
                        invoke    wsprintf, addr @szBuffer, addr szResData, \
                                eax, ecx, [ebx].Size1
                        invoke    _AppendInfo, addr @szBuffer
                .endif
                add     esi, sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
                pop     ecx
                dec     ecx
        .endw
;*****

```

在 `_ProcessPeFile` 子程序中，程序首先从数据目录的第 3 项得到资源数据块的入口地址，并用它来调用 `_ProcessRes` 子程序，这个子程序将递归调用自己来处理所有资源目录树上的节点。

_ProcessRes 子程序首先处理目录块中的第一个结构 IMAGE_RESOURCE_DIRECTORY，将结构中的 NumberOfNamedEntries 字段和 NumberOfIdEntries 字段相加得到后续的目录项总数，并构造一个循环来处理这些目录项。

```

.if      OffsetToData 字段的位 31=1
    (表明 OffsetToData 字段指向的是下一层的目录块)
    .if   当前是第 1 层
        (表明 Name1 字段代表的是资源类型)
        .if      Name1 字段的位 31=1
            Name1 指向的是一个 UNICODE 字符串

        .else
            Name1 中包含的是资源类型 ID

        .endif
    .elseif 当前是第 2 层
        (表明 Name1 字段代表的是资源名称)
        .if      Name1 字段的位 31=1
            Name1 指向的是一个 UNICODE 字符串

        .else
            Name1 中包含的是资源名称 ID

        .endif
    .endif
    将层次加 1 继续递归处理 OffsetToData 所指的下一层目录块
.else
    (表明 OffsetToData 字段指向的是 IMAGE_RESOURCE_DATA_ENTRY 结构)
    (表明 Name1 字段代表的是资源的代码页)
    IMAGE_RESOURCE_DATA_ENTRY 结构地址=OffsetToData 字段
    资源 RVA=IMAGE_RESOURCE_DATA_ENTRY.OffsetToData

```

```
资源大小=IMAGE_RESOURCE_DATA_ENTRY.Size1
#endif
```

例子代码在每次处理一个目录项或者资源数据的时候，都将它们的名称或 ID 等信息显示出来。如果例子中的代码被移植到其他地方用来寻找资源的话，这些显示信息的语句就可以全部去掉了，因为这时程序的最终目的就是最后两句获取资源 RVA 和大小的指令。

17.5 重定位表

什么是重定位，代码又是在什么情况下才需要重定位呢？这个问题早在 13.4.2 节中就回答过了，那就是在 32 位代码中，涉及直接寻址的指令都是需要重定位的（而在 DOS 的 16 位代码中，只有涉及段操作的指令才是需要重定位的，对此感兴趣的读者可以参考相关的资料），虽然 13.4.2 节的例子中给出了一段不需要重定位的代码，但这段代码的精髓在于将所有的直接寻址指令用寄存器寻址方式代替，如果这种方法成为操作系统处理重定位问题的标准办法，那就相当于不存在直接寻址指令了，这在编程中带来的麻烦是不可想象的，所以那种能自身完成重定位的代码只能在小范围内使用。

对于操作系统来说，其任务就是在对可执行程序透明的情况下完成重定位操作，在现实中，重定位信息是在编译的时候由编译器生成并被保留在可执行文件中的，在程序被执行前由操作系统根据重定位信息修正代码，这样在开发程序的时候就不用考虑重定位问题了。

重定位信息在 PE 文件中被存放在重定位表中，本节要讨论的就是重定位表的结构和使用方法。

17.5.1 重定位表的结构

1. 重定位所需的数据

在开始分析重定位表的结构之前需要了解两个问题：第一，对一条指令进行重定位需要哪些信息；第二，这些信息中哪些应该被保存在重定位表中。下面举例来说明这两个问题。

作为例子，现将 13.4.2 节中的那段代码搬回来：

```
:00400FFC 0000                                ;dwVar 变量
:00401000 55      push ebp
:00401001 8BEC    mov ebp, esp
:00401003 83C4FC    add esp, FFFFFFFC
:00401006 A1FC0F4000 mov eax, dword ptr [00400FFC] ;mov eax,dwVar
:0040100B 8B45FC    mov eax, dword ptr [ebp-04] ;mov eax,@dwLocal
:0040100E 8B4508    mov eax, dword ptr [ebp+08] ;mov eax,_dwParam
:00401011 C9      leave
:00401012 C20400    ret 0004
:00401015 68D2040000 push 000004D2
:0040101A E8E1FFFFFF call 00401000 ;invoke Proc1,1234
```

其中地址为 00401006h 处的 `mov eax, dword ptr [00400ffc]` 就是一句需要重定位的指令，当整个程序的起始地址位于 00400000h 处的时候，这句代码是正确的，假如将它移到 00500000h

处的时候，这句指令必须变成 `mov eax, dword ptr [00500ffc]` 才是正确的。这就意味着它需要重定位。

让我们看看需要改变的是什么，重定位前的指令机器码是 `A1 FC 0F 40 00`，而重定位后将是 `A1 FC 0F 50 00`，也就是说 `00401007h` 开始的双字 `00400ffc` 变成了 `00500ffc`，改变的正是起始地址的差值 (`00500000h - 00400000h`) = `00100000h`。

所以，重定位的算法可以描述为：将直接寻址指令中的双字地址加上模块实际装入地址与模块建议装入地址之差。为了进行这个运算，需要有 3 个数据，首先是需要修正的机器码地址；其次是模块的建议装入地址；最后是模块的实际装入地址。这就是第一个问题的答案。

在这 3 个数据中，模块的建议装入地址已经在 PE 文件头中定义了，而模块的实际装入地址是 Windows 装载器确定的，到装载文件的时候自然会知道，所以第二个问题的答案很简单，那就是应该被保存在重定位表中的仅仅是需要修正的代码的地址。

事实上正是如此，PE 文件的重定位表中保存的就是一大堆需要修正的代码的地址。

2. 重定位表的位置

重定位表一般会被单独存放在一个可丢弃的以 `“.reloc”` 命名的节中，但是和资源一样，这并不是必然的，因为重定位表放在其他节中也是合法的，惟一可以肯定的是，如果重定位表存在的话，它的地址肯定可以在 PE 文件头中的数据目录中找到。如表 17.4 所示，重定位表的位置和大小可以从数据目录中的第 6 个 `IMAGE_DATA_DIRECTORY` 结构中获取。

3. 重定位表的结构

虽然重定位表中的有用数据是那些需要重定位机器码的地址指针，但为了节省空间，PE 文件对存放的方式做了一些优化。

在正常的情况下，每个 32 位的指针占用 4 个字节，如果有 n 个重定位项，那么重定位表的总大小是 $4 \times n$ 字节大小。

直接寻址指令在程序中还是比较多的，在比较靠近的重定位表项中，32 位指针的高位地址总是相同的，如果把这些相近表项的高位地址统一表示，那么就可以省略一部分的空间，当按照一个内存页来分割时，在一个页面中寻址需要的指针位数是 12 位（一页等于 4096 字节，等于 2 的 12 次方），假如将这 12 位凑齐 16 位放入一个字类型的数据中，并用一个附加的双字来表示页的起始指针，另一个双字来表示本页中重定位项数的话，那么占用的总空间会是 $4 + 4 + 2 \times n$ 字节大小，计算一下就可以发现，当某个内存页中的重定位项多于 4 项的时候，后一种方法的占用空间就会比前面的方法要小。

PE 文件中重定位表的组织方法就是采用类似的按页分割的方法，从 PE 文件头的数据目录中得到重定位表的地址后，这个地址指向的就是顺序排列在一起的很多重定位块，每一块用来描述一个内存页中的所有重定位项。

每个重定位块以一个 `IMAGE_BASE_RELOCATION` 结构开头，后面跟着在本页面中使用的所有重定位项，每个重定位项占用 16 位的地址（也就是一个 word），结构的定义是这样的：

`IMAGE_BASE_RELOCATION` STRUCT

```

VirtualAddress dd ? ;重定位内存页的起始 RVA
SizeOfBlock dd ? ;重定位块的长度
IMAGE_BASE_RELOCATION ENDS

```

VirtualAddress 字段是当前页面起始地址的 RVA 值，本块中所有重定位项中的 12 位地址加上这个起始地址后就得到了真正的 RVA 值。SizeOfBlock 字段定义的是当前重定位块的大小，从这个字段的值可以算出块中重定位项的数量，由于 $\text{SizeOfBlock} = 4 + 4 + 2 \times n$ ，也就是 $\text{sizeof IMAGE_BASE_RELOCATION} + 2 \times n$ ，所以重定位项的数量 n 就等于 $(\text{SizeOfBlock} - \text{sizeof IMAGE_BASE_RELOCATION}) \div 2$ 。

IMAGE_BASE_RELOCATION 结构后面跟着的 n 个字就是重定位项，每个重定位项的 16 位数据位中的低 12 位就是需要重定位的数据在页面中的地址，剩下的高 4 位也没有被浪费，它们被用来描述当前重定位项的种类，其定义如表 17.7 所示。

表 17.7 重定位项高 4 位的含义

高 4 位值	在 Windows.inc 中的预定义值	含 义
0	IMAGE_REL_BASED_ABSOLUTE	这个重定位项无意义，仅仅用来作为对齐用
1	IMAGE_REL_BASED_HIGH	重定位地址指向的双字中，仅仅高 16 位需要被修正
2	IMAGE_REL_BASED_LOW	重定位地址指向的双字中，仅仅低 16 位需要被修正
3	IMAGE_REL_BASED_HIGHLOW	重定位地址指向的双字的 32 位都需要被修正，这是修正算法中举例的情况
4	IMAGE_REL_BASED_HIGHADJ	不详
5	IMAGE_REL_BASED_MIPS_JMPADDR	不详
6	IMAGE_REL_BASED_SECTION	不详
7	IMAGE_REL_BASED_REL32	不详

虽然高 4 位定义了多种重定位项的属性，但实际上在 PE 文件中只能看到 0 和 3 这两种情况。

所有的重定位块最终以一个 VirtualAddress 字段为 0 的 IMAGE_BASE_RELOCATION 结构作为结束，读者现在一定明白了为什么可执行文件的代码总是从装入地址的 1000h 处开始定义的了（比如，装入 00400000h 处的 .exe 文件的代码总是从 00401000h 开始，而装入 10000000h 处的 .dll 文件的代码总是从 10001000h 处开始），要是代码从装入地址处开始定义，那么第一页代码的重定位块的 VirtualAddress 字段就会是 0，这就和重定位块的结束方式冲突了。

下面的例子举出了一个重定位表的实际情况，假设模块被装入 00400000h 处：

重定位表偏移	数据	说明
0000h	00001000h	第一个块：页面起始地址是 00401000h
0004h	00000010h	重定位块长度是 10h
0008h	3012h	16 位重定位项，重定位位置：00401012h
000ah	3040h	16 位重定位项，重定位位置：00401040h
000ch	306fh	16 位重定位项，重定位位置：0040106fh
000eh	0000h	用于对齐的空白数据
0010h	00002000h	第二个块：页面起始地址是 00402000h
0014h	0000000ch	重定位块长度是 0ch
0018h	3080h	16 位重定位项，重定位位置：00402080h

17.5.2 查看 PE 文件的重定位表举例

```

mov     esi, eax
pop     eax
invoke  _GetRVASection, _lpFile, eax
invoke  sprintf, addr @szBuffer, addr szMsg, addr szFileName, eax
invoke  SetWindowText, hWndEdit, addr @szBuffer
assume  esi:ptr IMAGE_BASE_RELOCATION
;*****
; 循环处理每个重定位块
;*****
        .while [esi].VirtualAddress

```

17.6 应用实例

671

改及重组 PE 文件，另外，像 API Hook，PE 文件的内存映像 Dump 等应用则涉及分析内存中的 PE 映像。

本节将用另外的两个例子来进一步说明这些方面的应用，17.6.1 节将演示如何从内存中动态获取某个 API 的地址；17.6.2 节将演示如何在 PE 文件上添加一段可执行代码。

17.6.1 动态获取 API 入口地址

学习这个例子是为了加深对 PE 文件到内存的映射和使用导出表这两方面的知识的理解。

在 Win32 环境下编程，不使用 API 几乎是不可能的事情，一般情况下，在代码中使用 API 不外乎两种办法：第一是编译链接的时候使用导入库，那么生成的 PE 文件中就会包含导入表，这样程序执行时会由 Windows 装载器根据导入表中的信息来修正 API 调用语句中的地址；第二种方法是使用 LoadLibrary 函数动态装入某个 DLL 模块，并使用 GetProcAddress 函数从被装入的模块中获取 API 函数的地址。

假如有一段代码由于特殊的原因无法（或者实现的难度很大）在 PE 文件中使用导入表，比如，17.6.2 节中被加到其他 PE 文件上的代码或者在第 13 章中介绍的远程线程中运行的代码就是如此，在这种代码中，如何使用 API 函数呢？有人可能会说，那就用第二种办法好了！听起来不错，但这里有一个“先有鸡还是先有蛋”的问题，固然所有的 API 函数都可以用 LoadLibrary 函数和 GetProcAddress 函数配合来动态获取，但这两个函数本身也是 API，又怎样首先得到它们的地址呢？

本节的内容讲述如何用一种变通的办法来解决这个问题。

1. 原理

在 DOS 环境下，一个可执行文件既可以用 INT 21h/4ch 来结束程序，也可以用一个 Ret 指令来结束程序。实际上，在 Win32 下也可以用这种方法来结束程序，虽然大部分的 Win32 程序都使用 ExitProcess 函数来终止执行，但是使用 Ret 指令确实也是有效的。

如图 17.9 所示，当父进程要创建一个子进程的时候，它会调用 Kernel32.dll 中的 CreateProcess 函数，CreateProcess 函数在完成装载应用程序后，会将一个返回地址压入堆栈并转而执行应用程序，如果应用程序用 ExitProcess 函数来终止，那么这个返回地址没有什么用途，但如果应用程序使用 Ret 指令的话，程序就会返回 CreateProcess 函数设定的地址。也就是说，应用程序的主程序可以看做是被 Windows 调用的一个子程序。

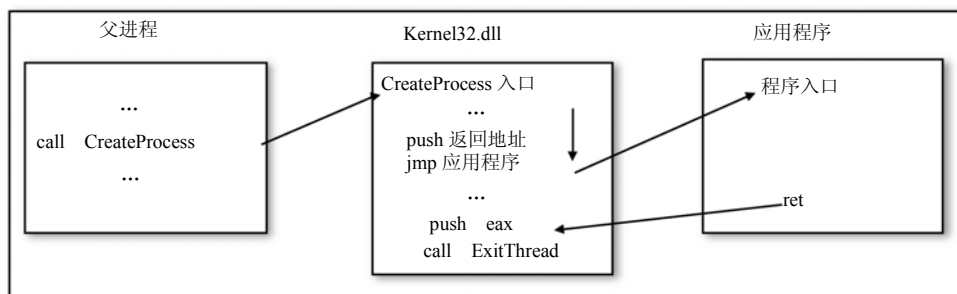


图 17.9 Win32 可执行文件退出的示意图

那么 Ret 指令返回到的地址上究竟有什么指令呢？用 Soft-ICE 看看就会发现，它包含一句 push eax 指令和一句 call ExitThread，也就是说，假如用 Ret 指令返回的话，Windows 会替程序去调用 ExitThread 函数，如果这是进程的最后一个线程的话，ExitThread 函数又会自动去调用 ExitProcess，这样程序就会被终止执行。

从这个过程可以得到一个很重要的数据，那就是堆栈中的返回地址，这个地址只要在程序入口的地方用[esp]就可以将它读出，说它重要是因为它位于 Kernel32.dll 模块中，而 LoadLibrary 和 GetProcAddress 函数正是处于 Kernel32.dll 模块中，换句话说就是，我们得到的地址和这两个函数近在咫尺，完全可以从这个地址经过某种算法来找到这两个函数的入口地址，得到这两个函数的入口地址以后，什么问题都解决了。

结合本章前面内容中提到过的两个事实，可以确定这种想法是可行的。

首先，PE 文件被装入内存后（包括 Kernel32.dll 文件），除了一些可丢弃的节如重定位节以外，其他的内容都会被装入内存，这样获取导出函数地址所需的 PE 文件头、导出表等数据都存在于内存中；其次，PE 文件被装入内存时是按 64k 对齐的，只要从 Ret 指令返回的地址按照对齐边界以 64k 为单位向低地址搜寻，就必然可以找到 Kernel32.dll 文件的文件头位置。

好了，有了 Kernel32.dll 的基址，接下来的事情就是按照 17.3.1 节的第 4 点所列的过程去操作了！

2. 例子

例子演示了上面所设想的功能,全部的源代码包含在本书所附光盘的 Chapter17\NoImport 目录中,主程序 NoImport.asm 的内容如下:

```

        .386
        .model flat,stdcall
        option casemap:none

;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
include        windows.inc

_ProtoGetProcAddress      typedef    proto      :dword, :dword
_ProtoLoadLibrary         typedef    proto      :dword
_ProtoMessageBox          typedef    proto      :dword, :dword, :dword, :dword
_ApiGetProcAddress        typedef    ptr        _ProtoGetProcAddress
_ApiLoadLibrary           typedef    ptr        _ProtoLoadLibrary
_ApiMessageBox            typedef    ptr        _ProtoMessageBox
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 数据段
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

        .data?

hDllKernel32      dd      ?
hDllUser32        dd      ?


_GetProcAddress   _ApiGetProcAddress?
_LoadLibrary      _ApiLoadLibrary      ?
_MessageBox       _ApiMessageBox       ?

        .const

```

这个程序的主要功能是用前面描述的方法获取 Kernel32.dll 的基址，并扫描导出表得到 GetProcAddress 函数的地址，然后调用这个函数得到 LoadLibrary 函数的地址。获得这两个关键函数的地址后，程序使用 LoadLibrary 函数装入 User32.dll 并获取 MessageBox 函数的地址，在显示了一个消息框以后用 Ret 指令结束程序。

674



哭

哭

676

```

; 计算 API 字符串的长度 (带尾部的 0)
;*****
        mov     edi, _lpszApi
        mov     ecx, -1
        xor     al, al
        cld
        repnz   scasb
        mov     ecx, edi
        sub     ecx, _lpszApi
        mov     @dwStringLength, ecx
;*****
; 从 PE 文件头的数据目录获取导出表地址
;*****
        mov     esi, _hModule
        add     esi, [esi + 3ch]
        assume  esi:ptr IMAGE_NT_HEADERS
        mov     esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
        add     esi, _hModule
        assume  esi:ptr IMAGE_EXPORT_DIRECTORY
;*****
; 查找符合名称的导出函数名
;*****
        mov     ebx, [esi].AddressOfNames
        add     ebx, _hModule
        xor     edx, edx
        .repeat
            push  esi
            mov   edi, [ebx]
            add   edi, _hModule
            mov   esi, _lpszApi
            mov   ecx, @dwStringLength
            repz  cmpsb
            .if   ZERO?
                pop  esi
                jmp  @F
            .endif
            pop  esi
            add   ebx, 4
            inc   edx
        .until  edx >= [esi].NumberOfNames
        jmp     _Error
@@:
;*****
; API 名称索引 --> 序号索引 --> 地址索引
;*****
        sub     ebx, [esi].AddressOfNames
        sub     ebx, _hModule
        shr     ebx, 1
        add     ebx, [esi].AddressOfNameOrdinals
        add     ebx, _hModule
        movzx   eax, word ptr [ebx]
        shl     eax, 2

```

文件中的_GetKernelBase 子程序的参数是主程序从堆栈中得到的返回地址,程序首先设置一个 SEH 异常处理子程序,以免在搜寻内存的过程中访问到无效的页面后出错;接下来将参数中传递过来的目标地址按 64K 对齐(与 0ffff0000h 进行 and 操作);然后以每次一个页的间隔在内存中寻找 DOS MZ 文件头标识和 PE 文件头标识,如果找到的话,表示这个页的起始地址就是 Kernel32.dll 模块的基址。

当调用_GetApi 子程序的时候，传递过来的 API 名称中不要忘了最后的“A”或“W”字符，比如 LoadLibrary 函数和 MessageBox 函数的真实函数名称根据版本的不同分别是“LoadLibraryA”，“MessageBoxA”或者“LoadLibraryW”和“MessageBoxW”，如果仅仅将“LoadLibrary”字符串传递过来的话，在导出表中是找不到这个函数的。

哭

哭

哭

哭

- 哭

哭

- 哭

哭

哭

哭

680


```

        invoke GlobalAlloc, GPTR, \
            [esi].OptionalHeader.SizeOfHeaders
        mov     @lpMemory, eax
        mov     edi, eax
        invoke RtlMoveMemory, edi, _lpFile, \
            [esi].OptionalHeader.SizeOfHeaders
        add     edi, esi
        sub     edi, _lpFile
        movzx   eax, [esi].FileHeader.NumberOfSections
        dec     eax
        mov     ecx, sizeof IMAGE_SECTION_HEADER
        mul     ecx

        mov     edx, edi
        add     edx, eax
        add     edx, sizeof IMAGE_NT_HEADERS
        mov     ebx, edx
        add     ebx, sizeof IMAGE_SECTION_HEADER
        assume  ebx:ptr IMAGE_SECTION_HEADER, edx:ptr IMAGE_SECTION_HEADER
;*****
; (Part 2.1) 检查是否有空闲的位置可供插入节表
;*****
        pushad
        mov     edi, ebx
        xor     eax, eax
        mov     ecx, IMAGE_SECTION_HEADER
        repz    scasb
        popad
        .if     ! ZERO?
;*****
; (Part 3.1) 如果没有新的节表空间的话, 则查看现存代码节的最后
; 是否存在足够的全零空间, 如果存在则在此处加入代码
;*****
        xor     eax, eax
        mov     ebx, edi
        add     ebx, sizeof IMAGE_NT_HEADERS
    .while  ax <= [esi].FileHeader.NumberOfSections
        mov     ecx, [ebx].SizeOfRawData
        .if     ecx && ([ebx].Characteristics & IMAGE_SCN_MEM_EXECUTE)
            sub     ecx, [ebx].Misc.VirtualSize
            .if     ecx > offset APPEND_CODE_END - offset APPEND_CODE
                or     [ebx].Characteristics, \
                    IMAGE_SCN_MEM_READ or IMAGE_SCN_MEM_WRITE
                jmp     @F
            .endif
        .endif
        add     ebx, IMAGE_SECTION_HEADER
        inc     ax
    .endw
;*****
        invoke CloseHandle, @hFile
        invoke DeleteFile, addr @szNewFile
        invoke SetWindowText, hWinEdit, addr szErrNoRoom
        jmp     _Ret

```

```

@@:
;*****
; 将新增代码加入代码节的空隙中
;*****
        mov     eax, [ebx].VirtualAddress
        add     eax, [ebx].Misc.VirtualSize
        mov     @dwAddCodeBase, eax
        mov     eax, [ebx].PointerToRawData
        add     eax, [ebx].Misc.VirtualSize
        mov     @dwAddCodeFile, eax
        add     [ebx].Misc.VirtualSize, \
                offset APPEND_CODE_END-offset APPEND_CODE
        invoke  SetFilePointer, @hFile, @dwAddCodeFile, \
                NULL, FILE_BEGIN
        mov     ecx, offset APPEND_CODE_END\
                -offset APPEND_CODE
        invoke  WriteFile, @hFile, offset APPEND_CODE, ecx, \
                addr @dwTemp, NULL
    .else
;*****
; (Part 3.2) 如果有新的节表空间的话, 加入一个新的节
;*****
        inc     [edi].FileHeader.NumberOfSections
        push    edx
        @@:
        mov     eax, [edx].PointerToRawData
;*****
; 当最后一个节是未初始化数据时, PointerToRawData 和 SizeOfRawData 等于 0
; 这时应该取前一个节的 PointerToRawData 和 SizeOfRawData 数据
;*****
        .if     ! eax
            sub     edx, sizeof IMAGE_SECTION_HEADER
            jmp     @B
        .endif
        add     eax, [edx].SizeOfRawData
        pop     edx
        mov     [ebx].PointerToRawData, eax
        mov     ecx, offset APPEND_CODE_END-\
                offset APPEND_CODE
        invoke  _Align, ecx, \
                [esi].OptionalHeader.FileAlignment
        mov     [ebx].SizeOfRawData, eax
        invoke  _Align, ecx, \
                [esi].OptionalHeader.SectionAlignment
        add     [edi].OptionalHeader.SizeOfCode, eax
        add     [edi].OptionalHeader.SizeOfImage, eax
        invoke  _Align, [edx].Misc.VirtualSize, \
                [esi].OptionalHeader.SectionAlignment
        add     eax, [edx].VirtualAddress
        mov     [ebx].VirtualAddress, eax
        mov     [ebx].Misc.VirtualSize, \
                offset APPEND_CODE_END-offset APPEND_CODE
        mov     [ebx].Characteristics, IMAGE_SCN_CNT_CODE\
                or IMAGE_SCN_MEM_EXECUTE\

```

```

                                or IMAGE_SCN_MEM_READ or IMAGE_SCN_MEM_WRITE
                                invoke  lstrcpy, addr [ebx].Name1, addr szMySection
;*****
; 将新增代码作为一个新的节写到文件尾部
;*****
                                invoke  SetFilePointer, @hFile, \
                                    [ebx].PointerToRawData, NULL, FILE_BEGIN
                                invoke  WriteFile, @hFile, offset APPEND_CODE, \
                                    [ebx].Misc.VirtualSize,      addr @dwTemp, NULL
                                mov      eax, [ebx].PointerToRawData
                                add      eax, [ebx].SizeOfRawData
                                invoke  SetFilePointer, @hFile, eax, NULL, FILE_BEGIN
                                invoke  SetEndOfFile, @hFile
;*****
                                push     [ebx].VirtualAddress      ;eax=新加代码的基地址
                                pop      @dwAddCodeBase
                                push     [ebx].PointerToRawData
                                pop      @dwAddCodeFile
                                .endif
;*****
; (Part 4) 修正文件入口指针并写入新的文件头
;*****
                                mov      eax, @dwAddCodeBase
                                add      eax, (offset _NewEntry-offset APPEND_CODE)
                                mov      [edi].OptionalHeader.AddressOfEntryPoint, eax
                                invoke  SetFilePointer, @hFile, 0, NULL, FILE_BEGIN
                                invoke  WriteFile, @hFile, @lpMemory, \
                                    [esi].OptionalHeader.SizeOfHeaders, addr @dwTemp, 0
;*****
; (Part 5) 修正新加代码中的 Jmp oldEntry 指令
;*****
                                push     [esi].OptionalHeader.AddressOfEntryPoint
                                pop      @dwEntry
                                mov      eax, @dwAddCodeBase
                                add      eax, (offset _ToOldEntry-offset APPEND_CODE+5)
                                sub      @dwEntry, eax
                                mov      ecx, @dwAddCodeFile
                                add      ecx, (offset _dwOldEntry-offset APPEND_CODE)
                                invoke  SetFilePointer, @hFile, ecx, NULL, FILE_BEGIN
                                invoke  WriteFile, @hFile, addr @dwEntry, 4, addr @dwTemp, NULL
;*****
; (Part 6) 关闭文件
;*****
                                invoke  GlobalFree, @lpMemory
                                invoke  CloseHandle, @hFile
                                invoke  wsprintf, addr @szBuffer, \
                                    Addr szSuccess, addr @szNewFile
                                invoke  SetWindowText, hWndEdit, addr @szBuffer
_Ret:
                                assume   esi:nothing
                                popad
                                ret
_ProcessPeFile  endp

```

另外一个源文件 `AddCode.asm` 中包含了被添加到目标 PE 文件中的代码，内容如下：

684

[illegible]

首先来看一下 AddCode.asm 中要被添加到其他可执行文件中的代码。这段代码是按照能够自身重定位的方式写的，而且必须按照这种格式书写，因为当它被添加到目标 PE 文件后，对于不同的 PE 文件所处的位置肯定是不同的，不进行重定位处理必然无法正常运行。

附加代码实现的功能和 17.6.1 节的 NoImport 例子大致相同，也是首先使用 17.6.1 节中的两个子程序获取 Kernel32.dll 模块的基址和 GetProcAddress 函数的入口地址，并由此最后得到 MessageBox 函数的入口地址以便显示消息框。

程序最后的_To0ldEntry 标号处的数据 0e9h 是 jmp xxxxxxxx 的机器码的第一个字节，它与下面的_dw0ldEntry 标号处的双字一起组成整个 jmp 指令，这条 jmp 指令将在主程序中根据具体情况修正。

好！现在来分析一下_ProcessPeFile.asm 文件中的代码。请读者注意源代码中的注释，注释将代码分为从 Part1 到 Part6 共 6 个部分。

Part 1 从原始 PE 文件拷贝一个名为“原始文件名_new.exe”的文件，这个文件将被添加上可执行代码，原来的“原始文件名.exe”文件则不会被改动。当文件成功拷贝后，程序将打

开拷贝生成的新文件以便进行修改。其中用到了 CopyFile 和 CreateFile 函数。

Part 2 的开始部分，程序使用 GlobalAlloc 函数分配一个等于目标 PE 文件的文件头大小的内存块，并使用 RtlMoveMemory 将 PE 文件头拷贝到这个内存块中，所有对 PE 文件头的修改操作都是在这个内存块中完成的，这个内存块的内容最终将被写到“原始文件名_new.exe”文件中。

完成拷贝工作以后，程序计算两个指针以备后用：指向节表最后一项的指针和指向节表尾部的指针，这两个指针可以从节表的数量和节表的长度计算而来，节表的数量是从 PE 文件头中的 FileHeader.NumberOfSections 字段获取的。

正如本节的开始所述，新增的代码既可以插入原代码所处的节的空隙中，也可以通过添加一个新的节来附在原文件的尾部。为了增加成功的机会，应该对这两种情况都予以考虑，于是程序在 Part 2.1 中对节表的尾部进行全零数据的扫描，如果存在一段全零的位置可供放入一个新的节表，那么采取增加新节的办法（Part 3.2），否则采用在代码节的空隙中插入的办法（Part 3.1）。

如下代码所示，Part3.1 中对所有节表进行循环扫描，以便于找到代码节并检测节的空隙是否可以容纳新增的代码，程序首先判断 SizeOfRawData 是否为 0，这个数值为 0 说明这个节是包含未初始化数据的节，不能用于插入代码，如果 SizeOfRawData 大于 0 的话，则检测 Characteristics 字段查看当前节是否为代码节（包含 IMAGE_SCN_MEM_EXECUTE 标志）。

```

; ebx 为某个节表的起始地址
mov     ecx, [ebx].SizeOfRawData
.if     ecx && ([ebx].Characteristics & IMAGE_SCN_MEM_EXECUTE)
    sub     ecx, [ebx].Misc.VirtualSize
    .if     ecx > offset APPEND_CODE_END - offset APPEND_CODE
        or      [ebx].Characteristics, \
                IMAGE_SCN_MEM_READ or IMAGE_SCN_MEM_WRITE
        add     [ebx].Misc.VirtualSize, \
                offset APPEND_CODE_END - offset APPEND_CODE
        jmp     将附加代码写入节的空隙中
    .endif
.endif
.endif

```

通过检测以后，程序计算空隙的大小（SizeOfRawData 和 Misc.VirtualSize 之差）是否大于插入代码的长度，如果空隙足够大的话，则进行插入操作。在插入代码的同时，这个节的属性中必须被加上 IMAGE_SCN_MEM_READ 和 IMAGE_SCN_MEM_WRITE 标志，因为附加代码中使用了对被加入部分进行写操作的指令。另外，VirtualSize 字段中的实际数据大小也需要被修正。

上面的程序只考虑了在代码节中插入的情况，要是代码节中的空隙大小不够，那么程序就退出了。实际上，程序也可以在其他节中插入代码，只要将节的属性同时也加上 IMAGE_SCN_MEM_EXECUTE 标志就可以了。要是进一步将程序完善的话，当单个节的空隙大小不够的时候，也可以将附加代码分块插入多个节中（CIH 病毒就是这么做的），不过这时附加代码中就必须考虑在执行前将代码重新拼装到一起这个步骤了。

当节表后面有多余空间的时候，程序采用增加新节区的办法来添加代码，这一部分由 Part 3.2 完成，程序首先在节表中加入一个新的节表项目，节表项中的 VirtualSize, VirtualAddress, SizeOfRawData, PointerToRawData, Characteristics 和 Name1 字段需要被设置。其中 Name1 中

的名称被设置为“.adata”；Characteristics 字段中的标志被设置为可执行和可读写，其他几个字段值的算法如下（下面的“上一节”指原始 PE 文件的最后一节）：

- $\text{PointerToRawData} = (\text{上一节的 PointerToRawData}) + (\text{上一节的 SizeOfRawData})$
- $\text{SizeOfRawData} = \text{附加代码的长度按 FileAlignment 值对齐}$
- $\text{VirtualAddress} = (\text{上一节的 VirtualAddress}) + (\text{上一节的 VirtualSize 按 SectionAlignment 的对齐值})$
- $\text{VirtualSize} = \text{附加代码的长度按 SectionAlignment 值对齐}$

其中的对齐算法是用_Align 子程序来完成的。

由于这种方法部分修改了文件的结构，所以必须同时对 PE 文件头中的相关字段进行调整，它们是 NumberOfSections、SizeOfCode 和 SizeOfImage 字段。如果 SizeOfImage 的值不被修正的话，Windows 将无法装入修改后的 PE 文件，系统会报“这不是一个有效的 Win32 可执行文件”的错误。Part 3.2 的最后，程序将附加代码写到文件的最后，由于附加代码的长度还没有按 FileAlignment 的值对齐，所以程序再次使用 SetFilePointer 函数将文件指针移动到对齐后的位置并用 SetFileEnd 函数将文件长度扩展到这里。

无论是 Part 3.1 还是 Part 3.2 的最后，程序将新增代码在文件中的位置和在内存中的位置分别保存在 @dwAddCodeFile 和 @dwAddCodeBase 变量中以备后用。

Part 4 修改 PE 文件头中的文件入口地址，并将修改后的整个 PE 文件头写入新文件中。

Part 5 将原始 PE 文件的入口地址取出，和附加代码的入口地址计算得出“jmp 原入口地址”这条指令中的二进制码值，并将这个值写到附加代码的对应位置中。jmp 指令的编码方式是由一个 0e9h 字节加上指令执行后的 EIP 的修正值，也就是说，当 jmp 指令的下一句指令地址是 addr1，而跳转的目标地址是 addr2 的话，那么 0e9h 字节后的双字的值就是 $\text{addr2} - \text{addr1}$ ，所以下面几句就是将指令改成“jmp 原入口地址”的样子：

	push	[esi].OptionalHeader.AddressOfEntryPoint
	pop	@dwEntry
	mov	eax, @dwAddCodeBase
(1)	add	eax, (offset _ToOldEntry - offset APPEND_CODE + 5)
(2)	sub	@dwEntry, eax

在指令序列执行前，esi 指向 PE 文件头，@dwAddCodeBase 中保存有新增代码被装载到内存后的起始地址，所以由 (1) 标出的指令执行后，eax 的值是 _ToOldEntry 后面 5 字节的位置，或者说是 jmp xxxxxxxx 后一条指令的位置，也就是上面算式中的 addr1。@dwEntry 中的原始值是可执行文件原来的入口地址，也即 addr2，指令 (2) 执行后，@dwEntry 中的值就是 $\text{addr2} - \text{addr1}$ 了，这就是需要填入 _dwOldEntry 位置的数据。

接下来程序用 SetFilePointer 函数将文件指针移动到新增代码中的 _dwOldEntry 位置，并将上面计算出的结果写入文件中。

Part 6 进行扫尾工作，如释放内存、关闭文件和显示成功信息等。至此，程序的所有功能就完成了。

第 18 章

ODBC 数据库编程

早在 1970 年, 关系型数据库的理论就已经被提出。在计算机软件的发展历史上, 数据库的发展和操作系统的发展几乎是同步的。在 MS-DOS 操作系统出现后的几年内, Oracle 公司就已经推出了第一个通用的数据库产品。世界十大软件厂商的排名中一直有数据库厂商的身影, 在 20 世纪 80 年代, 生产 dBase III 数据库的 Ashton-Tate 公司是全球第三大独立软件公司。现在, 生产 Oracle 数据库的 Oracle 公司是仅次于微软公司的全球第二大独立软件公司。

虽然汇编程序员平时的工作很少和数据库打交道, 但要知道如果能够在汇编语言中访问各种数据库, 那么就能显著地增加汇编语言的使用范围。本章将介绍如何使用 ODBC 接口访问各种类型的数据库, 并具体分析访问 Oracle、SQL Server 等主流数据库时的差异和注意事项。

18.1 基础知识

18.1.1 数据库接口的发展历史

在 DOS 时代, PC 机上流行的数据库是 dBase 数据库。现在, 市场上流行的数据库产品种类繁多, 大型的数据库产品有 Oracle, DB2, Sybase, Informix 和 SQL Server 等, 小型的数据库产品有 Access, MySQL, Foxpro 和 Paradox 等。

早期的数据库并没有提供开发接口, 比如, dBase 数据库的开发只能使用数据库自己的 IDE 环境和脚本语言, 开发完毕后, 在 dBase 软件中对脚本进行解释执行。要在 C 语言或者汇编语言中使用 dBase 数据库的话, 只能绕过 dBase 软件直接对 DBF 文件进行读写, 这样不但要对文件进行解码编码, 也因此放弃了数据库软件带来的数据查询、管理、统计等各种功能。

后期的数据库则提供了一些编程接口, 其中的一种是使用预编译器的 SQL 语句嵌入模式, 比如, Oracle 数据库的 Pro*C 和 SQL Server 的 ESQL。以 Pro*C 为例, 如果在 C 代码中访问 Oracle 数据库, 访问数据库的语句可以用伪码书写, Pro*C 预编译器会将这部分语句翻译成对 Oracle 接口 DLL 的调用语句, 然后与源代码中其余的标准 C 代码一并存成*.c 文件后, 再交给 C 编译器处理。另一种是 API 接口, 如 Oracle 的 OCI 接口, API 类型的编程接口比预编译方式要方便得多。这些由数据库产品提供的编程接口并没有统一的标准, 不同数据库的接口截然不同, 带

来的后果就是程序的通用性很差，一旦需要换一种数据库，原有的代码就完全无法使用。

为了能用同样的方法访问不同的数据库，一些软件厂商提出通用的数据库接口标准，如微软的 ODBC、DAO、RDO、OLE DB 和 ADO 接口，还有 Borland 公司的 BDE 接口等，这些接口对各类数据库的驱动程序进行了封装，应用程序只需通过标准的语法访问接口程序，接口程序会根据情况调用不同的驱动程序来访问相关的数据库。这样从应用程序的角度看，访问各种数据库（包括符合规范的未知数据库）的方法是一样的。

读者肯定或多或少听说过其中的几种接口，那么这么多的数据库接口有什么不同呢？又是哪种接口最适合于汇编语言使用呢？接下来我们来看看这两个问题的答案。

这要从数据库访问技术的演变过程谈起。当市场上出现众多的数据库产品之后，Borland 和微软都制定了数据库访问的规范，其中微软推出了 ODBC（Open DataBase Connectivity）规范，而 Borland 公司推出了 BDE（Borland Database Engine）规范。在推出的初期，BDE 的性能比 ODBC 要好，但随着微软对 ODBC 的改进，以及对操作系统的垄断，ODBC 标准最终占据了多数市场。到现在，BDE 只是 Borland 产品线上的数据库访问标准，只有 C++ Builder 和 Delphi 等开发工具在一直使用 BDE 接口。

微软的 ODBC 接口是作为一组 API 函数提供的，应用程序通过 API 访问 ODBC 接口，ODBC 接口再通过相关数据库的驱动程序访问数据库。当需要访问一种新的数据库时，只需使用新的数据库驱动程序替代旧的驱动程序，应用程序即可照常运行，而无须修改代码。

由于 ODBC 使用比较底层的 API 接口，凡是可以调用 API 的语言都可以使用 ODBC 接口，这样，C、VC++ 和 Win32 汇编等语言都能方便地使用 ODBC 接口，但是在 VB、ASP 中的 VBScript 等比较高级的语言中使用 API 却是非常麻烦的事情，所以微软设计了 DAO 接口来供这些语言访问数据库。

由于 DAO（Data Access Objects）是面向对象的接口，所以它可以很方便地在 VB 中使用，但 DAO 在内部通过 Jet 引擎来访问数据库，Jet 引擎本来是专门为访问 Access 数据库设计的。虽然用它访问 Access 数据库时很快也很有效，但访问非 Access 数据库时（如 SQL Server 和 Oracle），Jet 引擎将通过 ODBC 访问数据库（过程如图 18.1 所示）。所以，访问除了 Access 之外的数据库时，DAO 的速度比较慢。

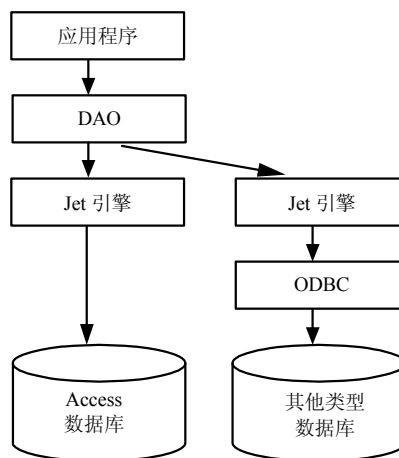


图 18.1 DAO 接口的架构过程图

为了克服这种缺陷，微软新设计了 RDO（Remote Data Objects）接口，与 DAO 类似，RDO 也是一个面向对象的接口，但 RDO 直接建立于 ODBC 之上而不是 Jet 引擎之上，不管访问何种类型的数据库，它都直接和 ODBC 接口对话，相当于在图 18.1 中去掉了 Jet 引擎这一层，从而使性能得到了提高。

ODBC、DAO 和 RDO 接口只能用来访问关系型数据库，但是应用程序也经常需要访问非关系

型的数据源，如 Microsoft Exchange Server 中的邮件、文本和图形、目录服务数据，甚至自定义的数据对象等。为此，微软设计了 OLE DB 接口。

OLE DB 接口也建立于 ODBC 之上，这个接口对关系型的数据库和非关系型的数据源提供了一致的访问。当访问关系型的数据库时，它仍然调用 ODBC 接口，对于非关系型的数据源，它使用与数据源相对应的驱动程序。这些驱动程序被称为 OLE DB Provider。

与 ODBC、DAO 和 RDO 接口相比，OLE DB 解决了非关系型数据的访问方式，使访问的数据类型得到了很大的扩展，但从编程的角度看，OLE DB 使用的是 COM 接口而不是 API 接口，COM 接口是一种二进制接口，调用的过程中大量地使用了指针变量，这对 C 和 VC++ 并不成问题，但由于 VB 和 VBScript 等高级语言不提供指针数据类型，所以无法直接调用 OLE DB。

所以，在此基础上微软又推出了另一个数据访问对象模型：ADO (ActiveX Data Objects)，由于 ADO 接口也是面向对象的，所以可以被 VB 使用。如图 18.2 所示，ADO 接口是以 OLE DB 为基础而封装的，它为 VB 等高级语言提供了 OLE DB 的所有功能。ADO 接口和 OLE DB 接口的关系就类似于 RDO 接口和 ODBC 接口的关系。

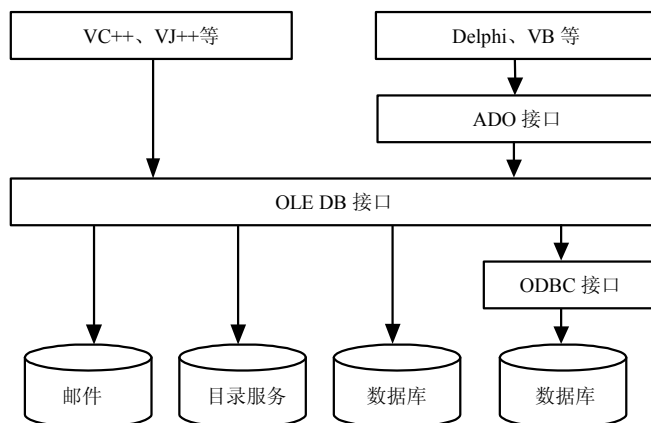


图 18.2 ADO 接口的架构图

看到这里，读者一定明白了第一个问题的答案。这些接口的差别在于层次、功能和接口封装方式的不同。具体使用哪种接口要看需要访问的数据类型，以及使用的语言能支持哪种接口封装方式，如 VB 程序访问关系型数据库时，可以使用 ADO、DAO 或者 RDO 接口，但是要访问非关系型的数据时，就只能使用 ADO 接口了。而 VC++ 程序既可以使用 ADO 接口，也可以使用 OLE DB 或者最底层的 ODBC 接口。

对于第二个问题，由于汇编语言是一种比较底层的语言，仅提供了对 API 的访问，虽然通过编程也能访问 COM 接口和 ActiveX 对象，但这属于舍近求远的使用方法，所以在 Win32 汇编中可选的是使用 ODBC 或者 BDE 接口。但是 BDE 不是 Windows 操作系统的内置组件，要想在某台计算机上运行使用了 BDE 接口的应用程序，就必须首先安装 BDE 环境，整个 BDE 环境的文件有 10MB 左右，所以从软件的兼容性和易用性考虑，使用 BDE 并不是一个明智的选择。ODBC 则是 Windows 98 及以上版本的标准组件，所以本书选择了 ODBC 作为数据库编程的接口 (Win95

系统需单独安装 ODBC 组件)。

现在再来详细看看 ODBC 接口的组成。ODBC 接口的架构如图 18.3 所示,当应用程序访问某种类型的数据库时,首先将相关的信息告诉 ODBC 管理器,这些信息包括使用的驱动程序名称,需要访问的数据库名称,用户名和密码等。ODBC 管理器根据这些信息选择合适的驱动程序连接到指定的数据库后,就可以使用 SQL 语句进行数据库操作了。

Windows 操作系统中附带了大量常见数据库的 ODBC 驱动程序,如 SQL Server, Access, Excel, Foxpro, Paradox, 以及 dBase 数据库的驱动程序等,由于 Oracle、Sybase 和 DB2 等大型数据库相对比较复杂,所以这些数据库的驱动程序并没有随操作系统提供,要访问这些数据库的话,必须首先在系统中安装这些数据库的客户端软件,以及对应的 ODBC 驱动程序后才能使用 ODBC 接口访问数据库。

因为本书的例子中访问的是 Access 数据库,所以读者在开始编程前并不需要安装额外的驱动程序,但如果读者要开发访问 Oracle 数据库的应用程序时,就必须首先安装 Oracle 客户端和 ODBC 驱动程序后才能进行程序的调试,开发出来的应用程序拷贝到其他计算机上运行时,该计算机上也必须安装 Oracle 客户端和 ODBC 驱动程序。

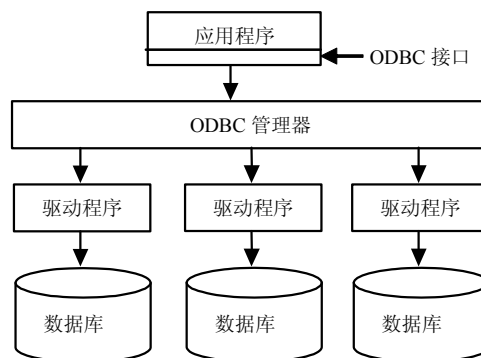


图 18.3 ODBC 接口的架构图

18.1.2 SQL 语言

1. 什么是 SQL 语言

结构化查询语言 (SQL /Structured Query Language) 是数据库系统的通用语言,利用它,用户可以用几乎同样的语句在不同的数据库系统中执行同样的操作。例如,不管是在 Oracle、SQL Server, 还是 Foxpro 数据库中,如果要从一张表中获取所有数据,使用的都是“select * from 表名”这样的语句。如果没有各数据库通用的 SQL 语言,实现 ODBC 这样的标准数据库接口是难以想像的。

SQL 是一个 ISO 官方标准,到现在为止,SQL 总共出了三代标准,除了最早的 SQL/86 标准之外,1992 年发布的标准被称为 SQL/2 或者 SQL/92 标准,最新的标准是 1999 年制定的 SQL/3 标准,也被称为 SQL/99 标准。

在 SQL 标准中,操作数据库的语句按照用途被分为下面四个大类:

- 数据查询语言 (DQL/Data query language) ——这部分只包含用于查询的 select 语句。
- 数据操作语言 (DML/Data manipulation language) ——这部分包含对数据进行维护的语句,如增加数据的 insert 语句、删除数据的 delete 语句和修改数据的 update 语句。

- 数据定义语言 (DDL/Data definition language) ——这部分包含对表结构、索引等各种对象进行维护的语句,如创建各种对象时用的 create 语句、修改对象的 alter 语句和删除对象的 drop 语句等。
- 数据控制语言 (DCL/Data control language) ——这部分包含和权限控制、事务控制等相关的语句。

SQL 标准是数据库语言中的官方语言,各种数据库均实现了 SQL 标准中的大部分语句,但是由于各种数据库的功能毕竟有很大的不同,所以除了“官方语言”之外,各种数据库中又存在少量的“方言”,这些方言反映了数据库对 SQL 标准的扩展,也反映了对 SQL 标准没有涉及的部分的处理方式。比如,同样是取数据库的日期,SQL Server 和 Sybase 中使用的是 getdate() 函数,Oracle 数据库中使用的是 sysdate 变量,而 Access 中使用的却是 now() 函数。再比如,Oracle 9i 数据库的 DML 语句中比 SQL/99 标准多了一个功能强大的 merge 语句。

要对某种数据库进行编程的时候,读者不仅要要对标准的 SQL 语句有所了解,而且对数据库中特定的语法也要有所了解,只有同时掌握了“官方语言”和“方言”,才能得心应手地发挥数据库的所有功能。当熟悉了某种数据库的编程后,一旦需要换一种数据库,那么只需对“方言”部分再进行熟悉就可以了。



本书已经假定读者对 SQL 语言有了相当的了解,所以不再对 SQL 语言的细节做介绍。如果读者对 SQL 语言的语法还不是很熟悉,请首先进行相关的学习。

2. 在 ODBC 中使用 SQL 语言

看到这里,读者脑海里一定有个问题,那就是:前面不是刚说过要用 Win32 汇编语言通过 ODBC 接口访问数据库吗,怎么后面又说操作数据库用的是 SQL 语言呢?

其实两者并不矛盾,操作数据库的确是用 SQL 语言,而汇编语言只不过是 SQL 语句以字符串的方式传递给数据库进行处理而已,传递时用的“通道”就是 ODBC 接口。其实,无论是用 ADO 接口还是 ODBC 接口,应用程序向数据库提交的命令都是一个 SQL 语句字符串,接口在这里起的都是应用程序和数据库之间的桥梁作用。

由于应用程序只管向 ODBC 接口发送 SQL 语句字符串,这个字符串是用“官方语言”写的还是以“方言”写的就没必要关心了,只要数据库能“听”懂就行,所以 SQL “方言”并不影响程序调用 ODBC 接口的方式,举例来说,要从 Oracle 数据库中获取时间,程序要向数据库发送“select sysdate from dual”字符串;而当数据库换成 Sybase 的时候,程序的汇编源代码并不需要有什么改变,只需将字符串改为“select getdate()”即可。

在 ODBC 程序的调试中,读者经常感到困惑的是出错原因定位的问题。比如,执行了一个查询语句后,却没有得到预期的数据,这时往往搞不清楚究竟是 SQL 语句写错了还是汇编代码写错了。知道了 ODBC 接口的作用后,这个问题就很好解决,那就是调试的时候必须首先保证 SQL 语句的正确性。

各种数据库都提供了附带的开发工具,如 Oracle 数据库中有 sqlplus,SQL Server 中有

“查询分析器”，在这些工具中可以输入 SQL 语句并进行执行，当需要在程序中使用某个 SQL 语句时，建议读者首先在这些开发工具中验证一下 SQL 语句的正确性，将 SQL 语句在开发工具中调试完毕后，再原封不动地拷贝到汇编源代码中定义成字符串，这样就可以专心调试汇编代码的错误。

另外，有些 SQL 语句在调试的时候没有发现错误，但随着程序的使用，在某些情况下却会出错。比如说，表中某个字段的长度定义为 10 个字节，大部分情况下执行 insert 语句都是成功的，但某些情况下出错却并不一定是汇编代码的错误，有可能是用户输入了超过 10 个字节的数据，造成 insert 语句因为字段内容太长而失败；也有可能是表中定义了惟一索引，插入的数据因为重复而失败。这时首先要做的事情是把执行失败的语句显示出来并重新拷贝到数据库的开发工具中去执行看看，是不是汇编代码的错误就能立即见分晓。

18.1.3 ODBC 程序的流程

如图 18.4 所示，使用 ODBC 接口访问数据库的操作总共分为 6 个步骤，它们分别是连接到数据库、初始化语句句柄、执行 SQL 语句、处理语句执行结果、事务控制，以及断开连接。根据具体的应用，部分步骤可以多次运行（如步骤 3 和步骤 4），而部分步骤可能被省略（如步骤 5）。在整个过程中需要用到近 20 个函数，这些函数的用法将在下面的内容中逐一介绍。

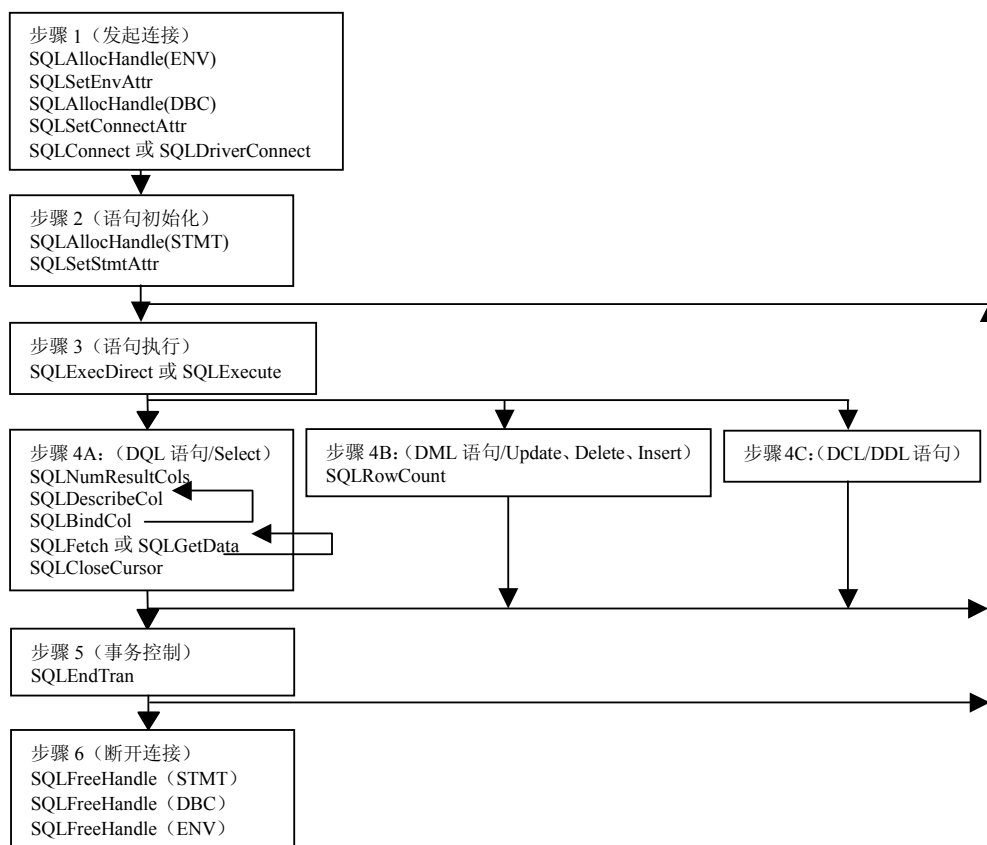


图 18.4 使用 ODBC 接口的流程

图中的步骤 1 和 6 将在 18.2 节中介绍, 步骤 3 到步骤 5 将在 18.3 节中介绍, 最后在 18.4 节中有一个综合的例子, 用来详细演示整个流程。

但是, 本章介绍的内容仅是 ODBC 接口的基本功能, 其他例如多记录集查询、利用光标修改记录、高性能光标等内容都没有涉及, 由于篇幅所限, 也没有列出所介绍函数的所有参数或返回代码的说明。如果读者有兴趣继续深入, 请参考 MSDN 或者 Microsoft Press 出版的《Microsoft ODBC 3.0 Programmer's Reference》。

18.2 连接数据库

18.2.1 连接和断开数据库

1. 分配环境句柄和连接句柄

ODBC 接口标准的最早版本是 1994 年提出的 (称为 ODBC 1.0 版), 到现在已经经过了多次的升级, 最新的版本是 ODBC 3.0, 后期的版本比前期版本增加了一些新的函数。为了程序的兼容性, 程序的开始部分需要首先指定使用哪个版本, 以便 ODBC 接口决定支持的函数集合, 这部分的工作即是环境的初始化工作。

环境的初始化工作只需进行一次, 如图 18.4 的步骤 1 所示, 这部分工作包括分配环境句柄、分配连接句柄, 以及对这些句柄进行一些适当的属性设置。

在 ODBC 3.0 版本之前, 分配环境句柄、连接句柄和语句句柄的函数是独立的, 它们分别是 SQLAllocEnv、SQLAllocConnect 和 SQLAllocStmt。但在 ODBC 3.0 版本中, 这些函数的功能全部由 SQLAllocHandle 函数来实现, 该函数的原型如下:

```
invoke    SQLAllocHandle, dwHandleType, dwInputHandle, lpOutputHandle
```

第一个参数 dwHandleType 用于指定需要分配的句柄类型, 取值可以是下面的常量之一:

- SQL_HANDLE_ENV——分配环境句柄 (Environment handle)
- SQL_HANDLE_DBC——分配连接句柄 (Connection handle)
- SQL_HANDLE_STMT——分配语句句柄 (Statement handle)
- SQL_HANDLE_DESC——分配描述符句柄 (Descriptor handle)

第二个参数 dwInputHandle 指定要分配句柄的“父”句柄, 分配语句句柄和描述符句柄时, 父句柄必须是连接句柄; 要分配连接句柄的话, 父句柄必须是环境句柄; 而环境句柄是最高层次的句柄, 所以分配环境句柄时该参数指定为 SQL_HANDLE_NULL。第三个参数 lpOutputHandle 是一个指针, 指向一个双字变量, 如果句柄分配成功的话, 函数会将句柄返回到这个双字中。

句柄分配成功后, 需要对句柄的各种属性进行适当的设置, 对环境句柄、连接句柄和语句句柄的属性进行设置的函数是不同的, 它们分别是 SQLSetEnvAttr、SQLSetConnectAttr 和 SQLSetStmtAttr 函数, 这三个函数的用法几乎一模一样:

```
invoke    SQLSet???Attr, hHandle, dwAttribute, ValuePtr, StringLength
```

函数名中的???代表 Env、Connect 或者 Stmt，函数的第一个参数 hHandle 是需要设置的句柄。第二个参数 dwAttribute 是一个常数，指定需要设置的属性类型，读者可以从 MSDN 中查看各种句柄可以设置的具体属性列表。

第三个参数 ValuePtr 和第四个参数 StringLength 的取值取决于属性类型，如果某个类型属性的取值是一个 32 位常数，那么 ValuePtr 就直接表示为该常数，而 StringLength 参数被忽略；如果属性的取值要用一个字符串或者一串二进制数据来表示，那么 ValuePtr 就被解释为指向字符串或二进制值的指针，这时 StringLength 参数表示字符串或者二进制数据的长度。

对于环境句柄，必须设置的属性值是将要使用的 ODBC 接口的版本号，这时的属性类型是 SQL_ATTR_ODBC_VERSION，要使用 3.0 版本接口的话，ValuePtr 取值为 SQL_OV_ODBC3。其他可选的属性值有用于连接池设置的 SQL_ATTR_CONNECTION_POOLING 和 SQL_ATTR_CP_MATCH 属性，具体的用法读者可以参考 MSDN。

对于连接句柄，有些属性必须在发起连接前进行设置，如 SQL_ATTR_LOGIN_TIMEOUT（连接的超时时间），有些则必须在连接成功后进行设置，如 SQL_ATTR_TRANSLATE_LIB 和 SQL_ATTR_TRANSLATE_OPTION（用于字符集转换）等，而大部分属性则可以在任何时刻进行设置，如 SQL_ATTR_ACCESS_MODE（连接的方式是只读还是读写）、SQL_ATTR_AUTOCOMMIT（事务的提交方式）等。在大多数情况下，我们没必要对连接句柄的属性进行设置，让其保持默认值即可。

环境句柄、连接句柄的分配和属性设置的代码举例如下，在完成这些步骤后，就可以用连接句柄来发起连接了：

```

.data?
hEnv    dd    ?    ; 环境句柄
hConn   dd    ?    ; 连接句柄

.code
invoke  SQLAllocHandle, SQL_HANDLE_ENV, SQL_NULL_HANDLE, addr hEnv
.if     ax != SQL_SUCCESS && ax != SQL_SUCCESS_WITH_INFO
    jmp  _Error
.endif
invoke  SQLSetEnvAttr, hEnv, SQL_ATTR_ODBC_VERSION, SQL_OV_ODBC3, 0
.if     ax != SQL_SUCCESS && ax != SQL_SUCCESS_WITH_INFO
    jmp  _Error
.endif
invoke  SQLAllocHandle, SQL_HANDLE_DBC, hEnv, addr hConn
.if     ax != SQL_SUCCESS && ax != SQL_SUCCESS_WITH_INFO
    jmp  _Error
.endif
...
;连接数据库，并执行 SQL 语句
...
_Error:
    出错处理

```

读者可以注意到，在上面的代码中，返回值的判断方法有些特别。其实代码中的方法正是绝大多数 ODBC 函数的返回值检测方法，这里有两点非常重要。

首先，所有 ODBC 函数返回值的类型是 SQLSMALLINT，也就是汇编中的 word 类型，所以判断 ODBC 函数是否执行成功，就要对 ax 而不是 eax 进行比较判断。

第二，ODBC 函数执行错误的话，返回的错误代码可能有很多种，但是表示成功的代码却有两种，其中 SQL_SUCCESS 表示函数执行成功，SQL_SUCCESS_WITH_INFO 表示函数执行成功但带有非致命的错误。返回这两种代码后，程序都应该按照执行成功进行处理。

无论函数的调用是成功还是失败，我们都可以通过调用 SQLGetDiagRec 或 SQLGetDiagField 函数来获得更多的信息，它们的用法和 Win32 API 中的 GetLastError 很相似。

2. 连接到数据库

一旦分配了环境句柄和连接句柄，并进行了适当的属性设置后，就可以调用 SQLConnect 或 SQLDriverConnect 函数来连接到数据库了。

要连接到数据库，就需要将与连接相关的信息告诉 ODBC 接口，如使用的驱动程序名称、数据库名称、登录数据库的用户名、密码，以及其他一些信息。有两种方法可以指定这些信息，那就是通过 DSN 或者在函数中直接指定这些信息。

DSN (Data Source Name/数据源) 其实就是上述信息的集合，比如，一个连接到 Access 数据库的数据源内容可能是这样的：

```
[ODBC]
DRIVER=Microsoft Access Driver (*.mdb)
UID=myusername
PWD=mypassword
DefaultDir=C:\Test
DBQ=C:\Test\Test.mdb
```

将这些信息保存并命名后，就是一个数据源。根据命名方式和保存位置的不同，可以将数据源分为三种：用户 DSN、系统 DSN 和文件 DSN。

因为用户 DSN 保存在注册表的 HKEY_CURRENT_USER\Software\ODBC\ODBC.INI 主键下，所以它只对当前登录用户可见，换一个用户登录到系统后，就没法看到这些 DSN 了（读者一定还记得第 15 章中介绍过 HKEY_CURRENT_USER 根键是根据不同用户映射到不同内容中去的）；而系统 DSN 则保存在注册表的 HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI 主键下，所以系统中所有的用户都可以看到它，包括 NT 中的服务都可以使用系统 DSN；文件 DSN 则将信息保存在一个单独的文件中，能访问到该文件，就可以使用文件中的信息。

DSN 可以在“控制面板”的“数据源 (ODBC)”栏目中创建，打开该栏目后，会弹出如图 18.5 所示的对话框，读者可以看到，对话框中有用户 DSN、系统 DSN、文件 DSN 等表单，每个表单中都有“添加”、“删除”和“配置”按钮。在对应的表单中单击“添加”就可以开始创建对应类型的 DSN。

创建 DSN 的过程如图 18.6 所示，单击“添加”按钮后，系统首先弹出图中 1 所示的对话框，让用户选择使用的驱动程序名称，选择完毕后，再由驱动程序弹出一个用于指定连接参数的对话框，由于各种数据库的连接参数命名方法各不相同，所以这时弹出的对话框也是各不相同的，如图中的 2 是选择 Oracle 数据库时的对话框，图中的 3 是指定了 SQL Server 数据库时的对话框，

而图中的 4 是指定了 Access 数据库时的对话框。在对话框中输入所需的信息（包括 DSN 的名称、数据库名、用户名和密码等）后再存盘，DSN 即创建完毕。



图 18.5 “控制面板”的“数据源”栏目

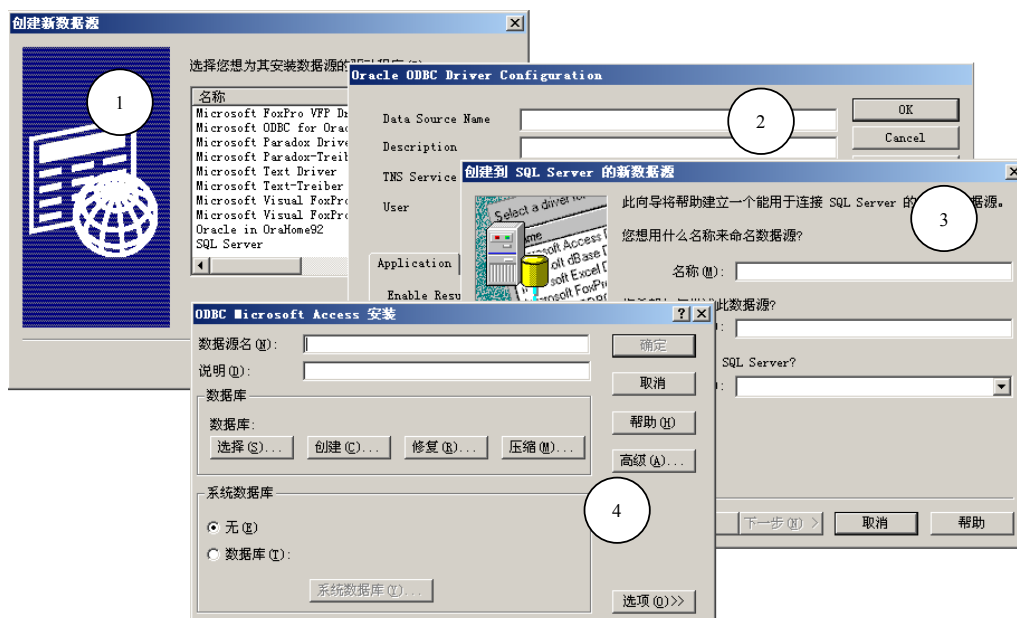


图 18.6 创建 DSN 的过程

假如创建的是用户 DSN 或者系统 DSN，那么现在我们就可以用 SQLConnect 函数来连接到数据库了（要用文件 DSN 的话，必须使用 SQLDriverConnect 函数）。SQLConnect 函数的用法非常简单：

```
invoke    SQLConnect, hConn, lpDSN, dwDSNLength, \
          lpUserName, dwNameLength, lpPassword, dwPasswordLength
```

其中 hConn 参数是在前面的例子中申请的连接句柄，lpDSN 指向一个包含 DSN 名称的字符串

串, dwDSNLength 指定为字符串的长度, 比如, DSN 的名称是“Test”时, 字符串就定义为“Test”, dwDSNLength 参数就是 4。

在创建 DSN 的时候, 用户可以直接在对话框中指定连接数据库使用的用户名和密码, 那么后面的四个参数可以全部指定为 NULL。有时为了安全起见, 在创建 DSN 的时候可以不指定用户名和密码, 这样就要在调用 SQLConnect 函数时在参数中指定这些信息, 这时 lpUserName 和 lpPassword 参数指向用户名和密码字符串, dwNameLength 和 dwPasswordLength 则是两个字符串的长度 (当然数据库可以匿名连接的时候, 即使 DSN 中没有保存用户名和密码, 在调用函数的时候这四个参数也可以指定为 NULL)。

SQLConnect 函数的最大缺点在于连接一个数据库之前必须创建它的 DSN。假如我们开发了一个应用程序并分发给用户后, 却要求每个用户在使用前必须在自己的计算机上创建指定的数据源, 那将是非常扫兴的事情。所以, 我们更多使用 SQLDriverConnect 函数来连接到数据库。

SQLDriverConnect 函数的用法相对复杂一点, 却可以通过参数直接指定所有和连接相关的信息, 所以在运行前不必先创建 DSN。该函数的用法如下:

invoke	SQLDriverConnect, hConn, hWnd, lpConnString, dwLength, lpOutConnString, \
	dwOutBufferSize, lpOutConnStringLength, dwDriverCompletion

hConn 是连接句柄。hWnd 是应用程序窗口的句柄, 因为函数可能会弹出一个模态对话框来要求用户输入一些信息, 该对话框会以 hWnd 作为父窗口句柄, 如果这个参数被置为 NULL, 那么对话框将没有父窗口句柄, 用户就可能在关闭对话框之前切换到主窗口中去。

lpConnString 是指向连接字符串的指针, 字符串中包含了连接到数据库的所有信息, 如驱动程序名称, 用户名和密码等。当连接到不同的数据库时, 连接字符串的格式是不同的, 但是不要着急, 下面的 18.2.2 节中马上会详细介绍连接字符串的写法, dwLength 参数则用来指定连接字符串的长度 (长度可以不包括字符串结尾的 0 字符)。

lpOutConnString 指向一个缓冲区, 如果函数执行成功的话, 将在缓冲区中返回一个完整的连接字符串。这听起来使人困惑, 连接字符串不是已经在前面的参数中提供了吗? 事实上, 我们提供的连接字符串可能会不完整, 这时, ODBC 驱动程序会提示用户输入更多信息, 并根据所有已知的信息, 以及默认的信息创建一个完整的连接字符串并将其放入缓冲区。即使我们提供的连接字符串已经可以工作了, 这个缓冲区中也会返回更详细的字符串。dwOutBufferSize 参数则用来指定缓冲区的长度。

lpOutConnStringLength 是一个指针, 指向一个双字变量, 函数在缓冲区中返回完整的连接字符串后, 在这个双字变量中返回字符串的长度。

最后一个参数 dwDriverCompletion 用来指示 ODBC 驱动程序是否提示用户输入更多信息。它可以是以下取值之一:

- SQL_DRIVER_PROMPT——ODBC 驱动程序将弹出一个对话框提示用户输入信息。驱动程序将利用这些信息来创建连接字符串。
- SQL_DRIVER_COMPLETE 或 SQL_DRIVER_COMPLETE_REQUIRED——仅当用户提供的连接

字符串不完全时，ODBC 驱动程序才会弹出对话框来提示用户输入缺少的信息。

- SQL_DRIVER_NOPROMPT——不管信息是否足够，ODBC 驱动程序都不会提示用户输入信息。

用 SQLDriverConnect 函数连接到数据库的例子如下：

```

strConn      .const
              db "DRIVER=Microsoft Access Driver (*.mdb);DBQ=c:\test.mdb",0
              .data?
szBuffer     db      1024 dup(?)
dwLength     dd      ?
              .code
              ..... ;前面例子中分配环境句柄和连接句柄的语句
              invoke SQLDriverConnect,hConn,hWnd,addr strConn,\
                  sizeof strConn,addr szBuffer,sizeof szBuffer,\
                  addr dwLength,SQL_DRIVER_COMPLETE
              .if    ax == SQL_SUCCESS || ax == SQL_SUCCESS_WITH_INFO
                  ; 连接成功
              .else
                  ; 连接失败
              .endif

```

不管是使用 SQLConnect 函数还是 SQLDriverConnect 函数，如果函数返回 SQL_SUCCESS 或者 SQL_SUCCESS_WITH_INFO，那就表示已经成功连接到数据库，接下来就可以执行 SQL 语句了。

3. 断开和数据库的连接

在成功连接到数据库后，我们就可以进行查询及其他操作了，这些将在 18.3 节中具体讨论，现在假设我们已完成了这些操作，那就需要断开数据库的连接并释放各种资源。

断开数据库使用 SQLDisconnect 函数，这个函数只需要一个参数：连接句柄：

```

invoke      SQLDisconnect, hConn

```

在断开连接后，还需要将连接句柄和环境句柄释放，这两个操作可以用 SQLFreeHandle 函数来完成。这个函数是 ODBC 3.0 版本提供的函数，在这以前，释放不同的句柄要分别由 SQLFreeConnect、SQLFreeEnv 及 SQLFreeStmt 函数来完成。SQLFreeHandle 函数的用法如下：

```

invoke      SQLFreeHandle,dwHandleType,hHandle

```

dwHandleType 是要释放的句柄类型，取值和申请时使用的值相同，也就是 SQL_HANDLE_DBC、SQL_HANDLE_ENV 或 SQL_HANDLE_STMT。hHandle 参数就是要释放的句柄。一般来说，扫尾工作的代码如下所示：

```

invoke      SQLDisconnect,hConn
invoke      SQLFreeHandle,SQL_HANDLE_DBC,hConn
invoke      SQLFreeHandle,SQL_HANDLE_ENV,hEnv

```

请注意上面代码的先后顺序。在释放环境句柄前，所有该句柄上的连接句柄必须首先被释放，否则释放环境句柄的 SQLFreeHandle 函数会返回 SQL_ERROR。同样，在释放连接句柄前，必须首先用 SQLDisconnect 函数将连接断开，否则 SQLFreeHandle 函数会返回 SQL_ERROR 并且

连接仍然被保持有效。



ODBC 规范要求 ODBC 驱动程序必须是多线程安全的，也就是说，应用程序可以在同一个环境句柄上申请多个连接句柄，然后在多个线程中，同时用这些连接句柄连接到不同的数据库（甚至是不同类型的数据库）。

另外，应用程序也可以在同一个连接句柄上申请多个语句句柄，然后在多个线程中，同时用这些语句句柄执行不同的 SQL 语句。

18.2.2 连接字符串

用 `SQLDriverConnect` 函数连接到数据库的时候，所有与连接数据库相关的信息都被放在连接字符串中，本节将具体讨论连接字符串的用法。

1. 连接字符串的格式

连接字符串的写法必须符合一定的格式，它由一系列的“属性名称=属性取值”字符串组合而成，每组属性定义之间用分号隔开：

属性 1=value;属性 2=value; ... ;属性 n=value

连接字符串中定义的属性分为两部分，其中的一部分是由 ODBC 管理器解释的，而另一部分则由驱动程序介绍。

由 ODBC 管理器解释的属性如下所示：

- DSN——使用系统 DSN 或者用户 DSN 来连接到数据库时，指定 DSN 的名称。
- FILEDSN——使用文件 DSN 来连接到数据库时，指定 DSN 的名称。
- DRIVER——使用指定的驱动程序来连接到数据库时，指定驱动程序的名称。

可以看到，在 `SQLDriverConnect` 函数中不仅可以使用系统 DSN 或用户 DSN 方式连接到数据库，也能使用文件 DSN，以及直接指定驱动程序方式，但是三者不能同时指定，只能选择其中的一种。

假如使用 DSN 方式连接到数据库，DSN 的名称是 `Test`，连接时使用的用户名和密码是 `myusername` 和 `mypassword`，那么连接字符串可以写成：

`szConn db "DSN=Test;UID=myusername;PWD=mypassword",0`

如果使用文件 DSN 方式连接到数据库，那么可以使用 `FILEDSN` 属性，假设文件 DSN 的文件名是 `Test.dsn`，连接字符串就可以写成：

`szConn db "FILEDSN=Test.dsn;UID=myusername;PWD=mypassword",0`

使用上面的两种方法前必须首先创建 DSN，比较麻烦，最方便的方法是直接指定驱动程序名称来连接到数据库，这就要用到 `DRIVER` 属性，假如使用的是 Access 数据库的驱动程序，那么连接字符串可以写成：

`szConn db "DRIVER=Microsoft Access Driver (*.mdb);",\`

"DBQ=C:\Test\Test.mdb;UID=myusername;PWD=mypassword",0

在字符串中, DRIVER 属性指定了驱动程序名称, DBQ 属性指定了数据库文件名。请注意 DRIVER 属性是由 ODBC 管理器解释的, 而其他的属性是由驱动程序解释的。

看到这里, 读者一定有个问题: DSN 是自己创建的, 使用的时候当然知道名称, 但是怎么知道驱动程序名称是什么呢? 其实驱动程序的列表可以在“控制面板”的“数据源(ODBC)”栏目中看到。如图 18.7 所示, 在“驱动程序”一栏中, 我们可以看到本机上安装的所有 ODBC 驱动程序。在安装了新的数据库后, 列表中会出现新数据库的驱动程序名称。要使用其中的某个驱动程序的时候, 只要将驱动程序名称抄过来就可以了。



图 18.7 本机上已安装的 ODBC 驱动程序

需要注意的是, 同一种数据库在不同的版本下驱动程序名称也可能不同, 如 Oracle 8i 版本的驱动名称是“Oracle ODBC Driver”, 但到 9i 版本时的驱动名称却变成了“Oracle in OraHome92”; Sybase 11 版本的驱动名称是“Sybase System 11”, 估计到 12 版的时候就会变成“Sybase System 12”了。

一个完善的应用程序应该考虑到这一点, 即使程序不打算支持多种数据库, 也应该具有通过参数设置来改变驱动程序名称的能力, 以便应对上述情况。



书写驱动程序名称的时候, 注意不要将名称中的空格忽略掉, 否则名称就无法匹配了。比如, 使用 Access 数据库驱动程序时, 名称中的“(*.mdb)”, 以及左括号前的一个空格都必须原封不动地抄回来, 否则连接时会因无法找到驱动程序而出错。

上面介绍的 DSN、FILEDSN、DRIVER 属性是由 ODBC 管理器解释的, ODBC 管理器在看到这些属性后, 才能找出对应的驱动程序并把其余的属性值传递给它。连接字符串中的其他属性则是原封不动地交给驱动程序解释的。

由驱动程序解释的属性值的定义就没这么统一了, 比如, 连接 Oracle 和 Access 数据库的时候, 数据库名称都可以用 DBQ 属性来指定; 但连接 Sybase 数据库时, 数据库却是由 SRVR 和 DB 两

个属性来共同指定的,其中 SRVR 指定数据库服务器的 IP 地址,DB 指定数据库的名称;到使用 SQL Server 数据库时,用于指定数据库名称的却是 SERVER 和 DATABASE 这两个属性,其中 SERVER 指定数据库服务器 IP 地址,DATABASE 属性指定数据库名称。

另外,各种数据库还有自己特定的属性值,比如,Oracle 数据库中用 MTS 属性来表示是否打开 Microsoft Transaction Server (MTS) 的支持,其他数据库中并没有 MTS 这样的模块,也就不会有 MTS 这样的属性值。

要获取连接字符串中各种数据库特有的属性值定义,最好的方法是查看各种数据库的 ODBC 驱动程序说明文档,一般来说,这些文档可以在数据库系统的帮助文件或用户手册中找到。

2. 连接到未知的数据库

虽然在数据库的帮助文档中可以找到 ODBC 连接字符串的定义方式,但还是有读者经常为连接字符串的写法感到困扰,因为在海量的帮助文档中找到合适的资料的确不是件容易的事情。实在无法找到帮助文档的时候,还有两种方法可以大致搞清楚驱动程序所需的属性值的定义方式。

第一个方法是利用 SQLDriverConnect 函数的 dwDriverCompletion 参数。我们知道,当 dwDriverCompletion 参数指定为 SQL_DRIVER_COMPLETE_REQUIRED 的时候,驱动程序会提示用户输入缺少的信息。这样,我们可以仅仅在连接字符串中指定驱动程序名称,然后在驱动程序弹出的对话框中输入相关的信息。如图 18.8 所示,不同数据库的驱动程序弹出的对话框有所不同(图中的 1 和 2 分别是 Oracle 和 SQL Server 数据库驱动程序弹出的提示框),但是根据上面的提示输入相关的信息并没有什么难度。

信息输入完毕并成功进行连接后,函数会在 lpOutConnString 指定的缓冲区中返回完整的连接字符串,将字符串中属性值的定义方式与在对话框中输入的信息进行比较,就可以发现各种属性的定义方法,以后就能根据我们的发现写出正确的连接字符串来了。上面的方法只能对比出一些必需的属性的定义方式,返回的连接字符串中还有很多定义了默认值的属性值,这些属性的含义就不得而知了。

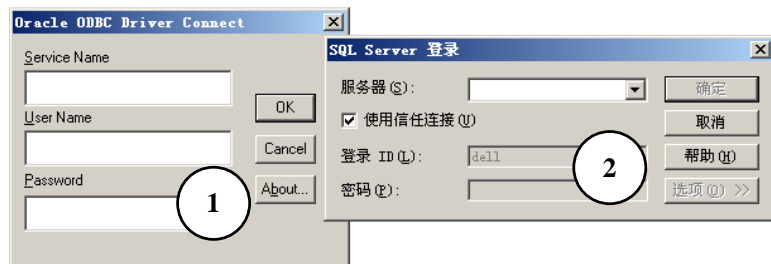


图 18.8 驱动程序的提示对话框

第二种方法则可以得到更详细的信息,我们可以利用 DSN 的创建对话框来输入信息(图 18.6 中所示),这时要输入的信息比图 18.8 中所示的对话框更加详细,DSN 创建完毕并命名为 xxx 后,在 SQLDriverConnect 函数中用“DSN=xxx”作为连接字符串,连接成功后,函数同样会在 lpOutConnString 指定的缓冲区中返回完整的连接字符串。将连接字符串的定义方式和在 DSN 的

创建对话框中输入的信息进行对比，就能得到更详细的属性定义方法。

有了这两种方法，就是遇到了未知的数据库驱动程序，我们仍然可以试出这种数据库的 ODBC 连接字符串的定义方式。

18.3 数据的管理

18.3.1 执行 SQL 语句

如图 18.4 中的第 2 步所示，连接建立后，我们就可以在这个连接上进行各种操作了，这时读者可以发挥自己的想象，用简单或复杂的 SQL 语句去完成各种工作。（在作者所处的部门中，经常有同事写一些单条语句长度达几个 KB 的 SQL 语句去完成很复杂的工作，这些 SQL 语句就被戏称为“惊天地泣鬼神”的 SQL 语句！）

1. 分配语句句柄

在让数据库执行 SQL 语句之前，必须首先分配一个语句（Statement）句柄并对它的属性进行适当的设置。分配语句句柄需要用到前面介绍过的 `SQLAllocHandle` 函数，`SQLSetStmtAttr` 函数则用于对句柄的属性进行适当的设置。下面是这个步骤的典型写法：

```

hStmt      .data?
            dd      ?
            .code
...          ;申请环境句柄，申请连接句柄，并连接到数据库的操作
invoke      SQLAllocHandle, SQL_HANDLE_STMT, hConn, addr hStmt
.if         ax != SQL_SUCCESS && ax != SQL_SUCCESS_WITH_INFO
            jmp      Error      ;错误处理
.endif
invoke      SQLSetStmtAttr, hStmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_STATIC, 0
...          ;执行 SQL 语句

```

一般仅对语句句柄的 `SQL_ATTR_CURSOR_TYPE` 属性（游标类型）进行设置，由于语句的游标默认是 `SQL_CURSOR_FORWARD_ONLY` 类型的，只能前移而不能随机移动，在使用中有诸多不便，所以一般将其设置为 `SQL_CURSOR_STATIC` 类型（静态游标，可以随机移动）。

2. 直接执行 SQL 语句

现在可以进行如图 18.3 所示的步骤——执行 SQL 语句了，通过语句句柄执行的 SQL 语句可以是 Select 语句，或者是 Insert、Delete 和 Update 等 DML 语句，也可以是“create table test(name char(10))”这样的创建表的语句。

实际上，凡是连接到的目标数据库的语法中支持的语句（也就是前面所称的“官方语言”和“方言”的总和），都可以通过语句句柄来执行，执行语句的函数仅起到“传声筒”的作用。比如，在 Oracle 数据库中用 call 语句来调用存储过程，假设现在 Oracle 数据库中建了一个 Proc1 存储过程，参数是一个字符串，那么连接到这个 Oracle 数据库后，就可以通过语句句柄执行“call Proc1('test')”这样的 SQL 语句。

有两个函数可以用来执行 SQL 语句，它们是 `SQLExecDirect` 和 `SQLExecute` 函数。前者直

接执行一个 SQL 语句，后者则执行编译过的语句，这样在多次调用同一个语句的时候效率比较高。

SQLExecDirect 函数的用法非常简单：

```
invoke SQLExecDirect, hStmt, lpSqlText, dwTextLength
```

hStmt 是语句句柄，lpSqlText 则是指向要执行的 SQL 语句字符串的指针，dwTextLength 参数是 SQL 语句字符串的长度（不包括字符串结尾的 0 字符）。

函数执行成功时，在 ax 中返回 SQL_SUCCESS_WITH_INFO 或者 SQL_SUCCESS，但有个例外，如果执行的 SQL 语句是 select 语句并且返回的结果集为空时，即使函数的执行是成功的，函数也会返回 SQL_NO_DATA 代码；如果执行失败，函数根据情况返回 SQL_ERROR 或 SQL_INVALID_HANDLE 等错误代码。

在应用中，如果多次执行类似于下面（1）和（2）所示的语句，一般先用（3）所示的字符串来定义 SQL 语句，然后在执行前用 sprintf 函数将具体参数代进去生成完整的语句，再用 SQLExecDirect 函数去执行：

```
(1) insert into address(id,name,phone) values(1,'张三','010-8888888')
(2) insert into address(id,name,phone) values(2,'李四','021-12345678')
(3) insert into address(id,name,phone) values(%d,'%s','%s')
```

这样，多次执行相似的语句时，从应用程序的角度来看，每次需要先用 sprintf 等字符串处理函数组合出完整的 SQL 语句；从数据库的角度来看，每次收到 SQL 语句时，都要经过语法分析、预编译等步骤后，才能开始执行。不管从哪个角度来看，用 SQLExecDirect 函数执行 SQL 语句的效率都是比较低的。

3. 编译执行 SQL 语句

如果需要多次执行相同（或者相似）的 SQL 语句，那就应该使用编译执行的方法。“相同”的语句指字面上一模一样的语句，包括语句中的参数都是相同的，“相似”的语句指语法相同但是语句中参数的值有所不同的语句，如上面例子中的（1）和（2）语句，这两条语句在语法分析和预编译的阶段完全可以用下面的语句来代替，只要在执行的时候将两个参数的值替换进去即可。

```
insert into address(id,name,phone) values(?, ?, ?)
```

按下面的步骤可以将 SQL 语句以先编译、后执行的方式执行：

（1）通过调用函数 SQLPrepare 来对语句进行语法分析和预编译操作。

（2）对于“相同”的语句，直接转第 3 步；对于“相似”的语句，用 SQLBindParameter 函数将参数的具体取值绑定到语句中。

（3）调用 SQLExecute 函数来执行语句。

一旦第 1 步成功完成后，第 2 步和第 3 步即可重复进行，这样从数据库的角度看，就不必每次对同样的 SQL 语句进行语法分析和预编译的操作，使效率大为提高；从应用程序的角度来看，不必每次使用字符串处理函数来组合 SQL 语句，也方便得多。

由于 SQLPrepare 函数与 SQLExecDirect 使用相同的三个参数，所以这里不再写出函数原型。而 SQLExecute 函数的用法更加简单，它只有语句句柄这样一个参数，比较复杂的是用于参数绑定的 SQLBindParameter 函数。

SQLBindParameter 函数的用法如下：

invoke	SQLBindParameter, hStmt, dwParamNumber, dwInputOutputType, \
	dwValueType, dwParamType, dwColumnSize, dwDecimalDigits, \
	lpParamValue, dwBufferLength, lpStrLenOrInd

每次调用 SQLBindParameter 函数可以将语句中的一个参数（也就是用问号表示的地方）绑定到一个缓冲区地址中，前面例句中有三个参数，就需要循环调用三次 SQLBindParameter 函数，绑定完毕后调用 SQLExecute 函数来执行语句的时候，ODBC 接口就会自动去绑定的缓冲区地址中取出数据，并代到 SQL 语句中去执行。

函数的 hStmt 参数是语句句柄，dwParamNumber 是要绑定的参数序号，序号从 1 开始计算，例如前面例子中的 SQL 语句有三个参数，那么三次调用 SQLBindParameter 函数时这个参数分别指定为 1、2、3。

dwInputOutputType 参数表明绑定的参数是用来输入还是输出的，“输入”是将参数中的值代到 SQL 语句中去执行，“输出”指函数将在操作结束时将结果放入参数中。大多数情况下，参数是以输入方式使用的，而输出参数经常用于存储过程执行后的结果返回。dwInputOutputType 参数的取值可以是下面三个数值之一：SQL_PARAM_INPUT_OUTPUT、SQL_PARAM_INPUT 和 SQL_PARAM_OUTPUT。

SQL 语句会对数据库中的某个表进行操作，接下来的三个参数是用来指定表中字段的定义的：其中 dwParamType 参数用来指定字段的类型，取值是 SQL 带头的常量（如表 18.1 所示），dwColumnSize 参数指定字段的长度，如果字段类型是数值型的话，dwDecimalDigits 参数指定小数部分的位数。假设上面例子中的 address 表是这样定义的：

```
create table address(
    id        integer,      // 4 字节的长整数
    name      char(10),    // 10 字节的字符型
    phone     char(50))    // 50 字节的字符型
```

那么绑定 id 字段时，这三个参数分别指定为 SQL_INTEGER、4、0，绑定 name 字段时，这三个参数分别指定为 SQL_CHAR、10、0。

函数的最后四个参数用来指定和缓冲区中数据相关的信息，其中 dwValueType 参数指定了缓冲区中存放数据的类型，可能的类型是一组以 SQL_C_开头的常数（如表 18.1 所示）。lpParamValue 参数是一个指向缓冲区的指针，dwBufferLength 参数表示缓冲区的长度，lpStrLenOrIndPtr 是指向一个双字的指针，双字中包含以下数值之一：

- 缓冲区中的数据实际长度。
- SQL_NTS——表示缓冲区中的数据是一个以 0 结尾的字符串（Null-Terminated String），这时由接口自动计算长度。

- SQL_NULL_DATA——表示参数取值为 NULL。
- SQL_DEFAULT_PARAM——表示参数取值为存储过程的默认值，它仅适用于已定义了默认参数值的存储过程。
- SQL_DATA_AT_EXEC——参数的数据将由 SQLPutData 传送，由于数据可能太大无法放入内存（比如，整个文件的数据），那么我们告诉 ODBC 驱动程序我们将用 SQLPutData 替代。

请注意 dwBufferLength 参数和 lpStrLenOrIndPtr 指向的双字中数据的区别，前者是缓冲区的最大长度，而后者是缓冲区中数据的实际长度。例如，绑定上面例子中的 name 字段时，由于 name 字段最长是 10 个字节的，为了能容纳字符串尾部的 0，我们可以分配一个 11 字节的缓冲区，这时 dwBufferLength 参数指定为 11；现在将字符串“张三”放入缓冲区，那么 lpStrLenOrIndPtr 指向的双字中的值指定为 4——这才是数据的实际字节数。

还是上面的语句为例，绑定参数的实际代码举例如下，注意为了简单起见，代码中忽略了对错误的判断：

```

szSQL    db      .const
           'insert into address(id,name,phone) values(?, ?, ?)', 0
           .data?
dwParam1 dd      ?      ;存放 id 字段的缓冲区
szParam2 db      11 dup (?)      ;存放 name 字段的缓冲区
szParam3 db      51 dup (?)      ;存放 phone 字段的缓冲区
dwSize1  dd      ?      ;存放 id 字段数据的长度
dwSize2  dd      ?      ;存放 name 字段数据的长度
dwSize3  dd      ?      ;存放 phone 字段数据的长度
           .code
           ...
           invoke  lstrlen, addr szSQL
           invoke  SQLPrepare, hStmt, addr szSQL, eax
           invoke  SQLBindParameter, hStmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, \
SQL_INTEGER, 4, 0, addr dwParam1, 4, addr dwSize1
           invoke  SQLBindParameter, hStmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, \
SQL_CHAR, 10, 0, addr szParam2, 11, addr dwSize2
           invoke  SQLBindParameter, hStmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR, \
SQL_CHAR, 50, 0, addr szParam3, 51, addr dwSize3
           ...

```

绑定参数完毕后，在 dwParam1、szParam2 和 szParam3 这三个缓冲区中分别放入数值 1，字符串“张三”和字符串“010-88888888”，在 dwSize1、dwSize2、dwSize3 中分别放入数值 4、4 和 12，然后再执行 SQLExecute 函数，即可插入第一条记录。

接下来将三个缓冲区中的数据换成数值 2、字符串“李四”和字符串“021-12345678”（由于三个数据的长度刚好和前面的相同，所以就不必改变 dwSize1 等的取值了），再执行 SQLExecute 函数，即可插入第二条记录。如果还要插入成千上万条记录，只要重复这个步骤就可以了。

💡 多次执行相同（或相似）的 SQL 语句时，采用编译执行的方法效率很高，但对于只执行一次或数次的 SQL 语句来说，编译执行并没有优势，这时使用直接执行更为方便。另外，编译执行比较适合应用程序内置的 SQL 语句，如果执行的是用户实时输入的 SQL 语句，那么只能使用直接执行的方式了。

5. 参数数据类型的转换

现在回过头来思考一个问题：dwValueType 参数是不是有点多余呢？因为 dwParamType 参数中已经指定了字段的数据类型，只要在缓冲区中放入和字段同类型的数据就可以了呀，为什么非要用 dwValueType 参数重新指定一次数据类型呢？

其实这个参数并不多余，当缓冲区中的数据类型和字段数据类型是一致的情况下，这两个参数的确是重复的，但是当两者指定的数据类型不一致时，ODBC 接口可以自动将数据进行转换后再交给数据库进行处理，这样可以带来很多方便。

上面的例子中，假如原始 id 数据是字符串类型的，但是数据库表中的对应字段却是整数型的，那么我们可以将转换的工作交给接口去完成，这时只需在绑定时将 dwValueType 参数指定为 SQL_C_CHAR 类型即可：

```
invoke    SQLBindParameter, hStmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, \
          SQL_INTEGER, 4, 0, addr szParam1, 10, addr dwSize1
```

然后，在绑定到 id 字段的缓冲区 szParam1 中放入字符串“1”，在 dwSize1 中放入字符串的长度 1，函数会自动将其转换成数值 1 后再交给数据库处理。

缓冲区中的数据类型在 ODBC 接口中以一系列的 SQL_C_开头的常量来定义（称为 C 数据类型），而数据库表中的字段类型则以 SQL_开头的常量来定义（称为 SQL 数据类型），两者的对应关系以及在汇编中的定义方式如表 18.1 所示，当绑定时 dwParamType 参数和 dwValueType 参数指定的类型一致时，接口不进行数据类型转换，不一致时则自动进行转换。

表 18.1 ODBC 接口中部分数据类型的对应关系

表中字段的类型	SQL 数据类型	C 数据类型	汇编语言类型
CHAR(n)	SQL_CHAR	SQL_C_CHAR	n 个 db
VARCHAR(n)	SQL_VARCHAR	SQL_C_CHAR	n 个 db
NUMERIC(p, s)	SQL_NUMERIC	SQL_C_NUMERIC	由 1 字节的 scale、1 字节的 sign 和 SQL_MAX_NUMERIC_LEN 字节的 val 字段组成
SMALLINT	SQL_SMALLINT	SQL_C_SSHORT	sword
INTEGER	SQL_INTEGER	SQL_C_SLONG 或 SQL_C_ULONG	sdword 或 dword
REAL	SQL_REAL	SQL_C_FLOAT	real4

续表

表中字段的类型	SQL 数据类型	C 数据类型	汇编语言类型
FLOAT(p)	SQL_FLOAT	根据 p 的大小用 SQL_C_FLOAT 或 SQL_C_DOUBLE	根据 p 的大小用 real4 或 real8

DOUBLE PRECISION	SQL_DOUBLE	SQL_C_DOUBLE	real8
BIGINT	SQL_BIGINT	SQL_C_SBIGINT	qword
BINARY (n)	SQL_BINARY	SQL_C_BINARY	n 个 db
DATE	SQL_TYPE_DATE	SQL_C_TYPE_DATE	由 2 字节的 year，4 字节的 month 和 4 字节的 day 组成
TIME	SQL_TYPE_TIME	SQL_C_TYPE_TIME	由 2 字节的 hour，2 字节的 minute 和 2 字节的 second 组成

表中列出的只是最常用的部分数据类型，全部数据类型的列表，以及不同数据类型之间转换的注意事项请参考 MSDN 中的 ODBC 部分。

在汇编语言中，遇到整数类型的字段时，一般将缓冲区中的整数指定为 SQL_C_ULONG 类型，遇到字符串类型的字段时，将缓冲区中的字符串数据指定为 SQL_C_CHAR 类型。除此之外，通常将数据以字符串方式表现并指定为 SQL_C_CHAR 类型，然后由接口进行转换。比如表示浮点数时用“1.23”类型的字符串，表示日期时用“2005-01-01”类型的字符串，这样可以避免一些因不了解数据存储格式而造成的错误。

5. 重新绑定参数

如果想用同样的语句句柄来执行新的语句，那么没必要将其关闭并重新分配一个语句句柄，只需用 SQL_RESET_PARAMS 参数来调用 SQLFreeStmt 函数来解除与参数的绑定就可以了：

invoke	SQLFreeStmt, hStmt, SQL_RESET_PARAMS
--------	--------------------------------------

接下来就可以使用原来的语句句柄来执行新的 SQL 语句了。

18.3.2 执行结果的处理

SQL 语句执行完毕后，程序需要对执行的结果进行处理，不同类型语句的结果处理方式是不同的，DQL (select) 语句执行后需要获取查询结果，DML (insert/update/delete) 语句需要获取语句修改的记录行数，而 DDL 及 DCL (创建或删除表、索引等对象，以及权限控制) 等语句只需检测语句是否执行成功即可。在大部分的情况下，程序中执行的是预定义的 SQL 语句，也就是说写程序的时候已经知道 SQL 语句的类型，所以在执行语句后可以直接转到相关的处理流程中去。

当执行的是 DDL、DCL 等语句的时候（也就是图 18.4 中的步骤 4C），这时只需检测 SQLExecDirect 或 SQLExecute 函数的返回值是否是 SQL_SUCCESS_WITH_INFO 或 SQL_SUCCESS 即可，如果是则表示语句执行成功，否则表示语句执行失败。

当执行的是 DML 语句时（也就是图 18.4 中的步骤 4B），如果语句执行成功，则可以用 SQLRowCount 函数获取语句修改的记录行数，这样可以判断语句的执行是否符合预期的结果，比如希望删除 1 行记录，虽然 delete 语句执行成功，但通过 SQLRowCount 函数发现删除掉的却有 10 行时，就不是预期的结果了，SQLRowCount 函数的用法是：

invoke	SQLRowCount, hStmt, lpdwRows
--------	------------------------------

hStmt 参数是执行了 SQL 语句的语句句柄，lpdwRows 参数是指向一个双字的指针，函数将

在这个双字中返回 insert/update/delete 语句修改的记录行数。如果执行的不是 DML 语句，那么函数在双字中返回-1（根据 MSDN 文档，这种情况下返回值由数据库的驱动程序自定义，但迄今为止，所有数据库的 ODBC 驱动程序返回的值都是-1）。

当执行的是 select 语句时，我们就有很多事情要做了，将整个结果集的数据取回来是件比较复杂的事情，这个问题将留到下一节再详细讨论。在这里先看看另一个经常遇到的问题：虽然大部分的情况程序中执行的是预定义的 SQL 语句，但在有些程序中，程序执行的是用户自由输入的 SQL 语句（比如 18.4 节中的例子程序），这时我们怎么知道该进入哪种流程进行处理呢？

当然，读者可能会说，检测一下输入的 SQL 语句的第一个单词不就行了吗？当然不行，比如，检测到 insert 单词的时候，当然可以认为是遇到 DML 语句了，但是 Oracle 9i 数据库的“方言”中，merge 语句也是 DML 语句，以后也不会排除各种数据库的“方言”中出现各种新的语句来，所以靠分析关键字是不准确的。

这时，我们要用到 SQLNumResultCols 函数，并将这个函数和 SQLRowCount 等函数组合起来进行判断即可。

SQLNumResultCols 函数用于获取 SQL 语句执行后返回的结果集的列数，比如，执行了“select id,name from address”，由于结果集中有 id 和 name 两个列，那么我们会得到 2；而执行的是“select * from address”时，表中定义了多少列，就会得到对应的列数。

SQLNumResultCols 函数的用法是：

dwRecordCols	. data? dd	?
	. code invoke	SQLNumResultCols, hStmt, addr dwRecordCols
	and	dwRecordCols, 0ffffh

函数的第一个参数 hStmt 是执行了 SQL 语句的语句句柄，第二个参数是指向一个 word 的指针，函数执行后，在这个 word 中返回结果集中的列数；如果执行的不是 select 语句，那么显然语句不会产生结果集，这时在 word 中返回的是 0。为了在其他语句中方便使用，一般将第二个参数指向的变量定义成一个 dword 而不是 word，这时要注意在函数执行后将 dword 的高 16 位清零，否则容易造成不必要的错误（如上面例子中的代码）。

如果用 SQLNumResultCols 函数得到的列数大于 0，那么执行的肯定是 select 语句；列数等于 0 则表示执行的不是 select 语句，这时要用 SQLRowCount 函数继续进行判断。



请读者注意区分没有结果集和结果集中没有数据的情况，非 select 语句执行后，结果是“没有结果集”；而 select 语句执行后没有得到任何符合条件的记录时（也就是返回的记录是 0 条），称为“结果集中没有数据”，这时 SQLNumResultCols 函数返回的还是正确的列数。

结合上面几个函数的功能，可以动态判断执行的语句类型，这在执行了用户自由输入的 SQL 语句后就显得非常有用：

```

.data?
hStmt    dd    ?
dwCols   dd    ?
dwRows   dd    ?
.code
invoke    SQLExecDirect, hStmt, addr szSQL, sizeof szSQL
.if       ax != SQL_SUCCESS && ax != SQL_SUCCESS_WITH_INFO && ax != SQL_NO_DATA
    jmp    Error    ; 语句执行失败
.endif
invoke    SQLNumResultCols, hStmt, addr dwCols
and       dwCols, 0ffffh
.if       dwCols
    ...           ; 执行的是 select 语句, 在这里获取结果集中的数据
.else
    invoke SQLRowCount, hStmt, addr dwRows
    .if     dwRows == -1
        ...     ; 执行的是 DDL 或 DCL 语句
    .else
        ...     ; 执行的是 DML 语句
    .endif
.endif
.endif

```

在执行 `SQLExecDirect` 函数后, 注意对返回值要检测 `SQL_NO_DATA`, 因为这时也表示函数是执行成功的。然后执行 `SQLNumResultCols` 函数, 如果得到的结果集列数大于 0, 则表示执行的是 `select` 语句, 这时可以进行获取结果集中的数据的操作了。

如果得到的结果集列数为 0, 则再执行 `SQLRowCount` 函数来检测语句影响的行数, 得到的结果不是 -1, 则表示执行的是 DML 语句; 是 -1 的话就表示执行的是 DDL 或 DCL 语句了。

18.3.3 获取结果集中的数据

当成功执行了 `select` 语句时, 下一步就是将查询到的结果集提取出来, “结果集”这个名词在前面已经多次提及, 那么究竟什么是结果集呢?

我们知道, `select` 语句从表中返回一些数据, 这些数据可能由多条记录组成, 每条记录中可能有多个字段。如图 18.9 所示, 一般将返回结果的集合称为结果集 (RecordSet), 结果集中的一条记录称为行 (Row), 结果集中的某个字段的集合称为列 (Column), 一行中某个列的内容 (也就是行和列的交叉点) 就称为字段 (Field)。另外, 由于程序一般以每次一行为单位访问结果集中的数据, 所以语句句柄中有一个指示当前行位置的指针 (就像文件句柄中的当前位置指针一样), 这个指针被称为游标 (Cursor), 当执行的是 `select` 语句时, 游标默认被打开, 并且位于结果集的第一行之前。注意游标不是指向第一行, 第一行的行号为 1, 第一行之前则称为 BOF/Begin of File, 而最后一行之后称为 EOF/End of File。

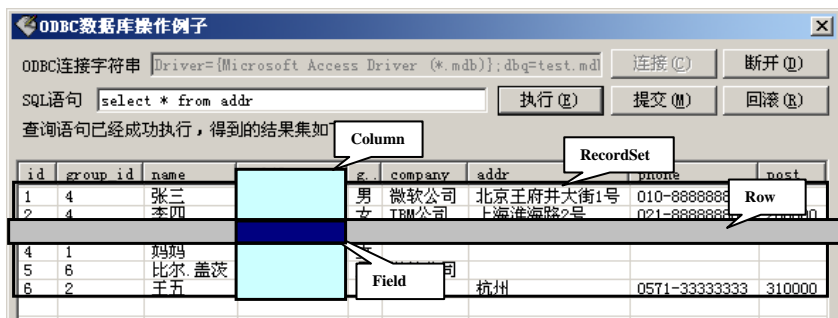


图 18.9 结果集、行、列和字段

游标的类型和读取记录的方式有很大关系，默认情况下，游标只能在结果集中向前移动（SQL_CURSOR_FORWARD_ONLY 类型），若想回到上一行，则必须先关闭游标然后重新打开（即重新执行语句），然后再从结果集的开始移动直到到达需要的行。静态游标则是可滚动的，它可以随机访问结果集，如果需要随机访问记录，则必须在语句执行前将语句的 SQL_ATTR_CURSOR_TYPE 属性设置为 SQL_CURSOR_STATIC。

1. 列缓冲区的绑定

如图 18.4 的步骤 4A 所示，获取结果集中数据的步骤是用多次调用 SQLBindCol 函数将结果集中的每个列绑定到指定的缓冲区地址，然后使用 SQLFetch 或 SQLFetchScroll 函数将当前行中每个列的数据传递到绑定的缓冲区中，重复调用 SQLFetch 函数，就能一行一行地取回数据，直到取出所有的行为止。在所有数据获取取完后，用 SQLCloseCursor 函数关闭游标后，即可将结果集释放，语句句柄就能用来重新执行其他语句。

SQLBindCol 函数的用法是：

```
invoke    SQLBindCol, hStmt, dwColNumber, dwTargetType, lpTargetValue, \
          dwBufferLength, lpStrLenOrInd
```

函数的第一个参数 hStmt 是执行了 SQL 语句的语句句柄，dwColNumber 是结果集中要绑定的列序号，序号从 1 开始计数，比如，对于“select id, name from address”语句返回的结果集，绑定 id 列的时候序号指定为 1，绑定 name 列的时候序号指定为 2。

dwTargetType 参数指定了传送至缓冲区中的数据的数据类型，类型可以是表 18.1 中的 C 数据类型（也就是以 SQL_C_ 开头的预定义值）。lpTargetValue 参数则是指向缓冲区的指针，以后调用 SQLFetch 函数来获取数据时，函数会将列的类型转换到符合 dwTargetType 参数定义的格式后再放入缓冲区。例如，id 字段是整数型的，结果集中返回的数据是 100，当缓冲区类型指定为 SQL_C_ULONG 时，那么放到缓冲区中的会是双字 100，而缓冲区类型指定为 SQL_C_CHAR 时，放到缓冲区中的就会是字符串“100”和一个结尾的 0 字符。

dwBufferLength 参数是由 lpTargetValue 指向的缓冲区的长度，lpStrLenOrInd 指向一个双字，以后用 SQLFetch 函数获取数据的时候，函数会在双字中返回数据的实际长度信息，具体的返回值同 SQLBindParameter 函数的同名参数的定义。



用 SQLFetch 函数获取数据的时候,有必要检查 lpStrLenOrInd 指向的双字中的返回值,假如返回值是 SQL_NULL_DATA 的话,表示列数据为空,这时函数不会改写列缓冲区。如果这时直接去读缓冲区,可能会得到上一次调用 SQLFetch 留下的数据。

读者还应该理解 NULL 和空字符串之间的区别,以一个字符串指针来类比,空字符串相当于一个有效的指针已经指向了字符串缓冲区,但缓冲区中只有一个 0 字符;而 NULL 相当于指针的值就是 0,指针本身就是无效的。

一次调用 SQLBindCol 函数只能绑定一个列,如果结果集中有两个列,那就需要调用两次 SQLBindCol 函数。但是绑定完毕以后,以后每次调用 SQLFetch 函数,列数据都会传递到指定的缓冲区中,这和 SQLBindParameter 函数的工作方式类似,用 SQLBindParameter 绑定输入参数缓冲区的操作也只需进行一次,以后每次调用 SQLExecute 函数,函数都会去同样的缓冲区中取数据。

当然,我们也可以只绑定结果集中部分的列,那么以后调用 SQLFetch 函数时就只会返回已绑定的列的数据,其余未绑定的列将被忽略。

当需要取消绑定的时候,只需用 SQL_UNBIND 参数来调用 SQLFreeStmt 函数即可:

invoke	SQLFreeStmt, hStmt, SQL_UNBIND
--------	--------------------------------

2. 动态检测列的属性并绑定缓冲区

当执行的 select 语句是程序中预定义的语句时,程序已经知道结果集中会有多少个列,以及都是哪些列,也肯定知道列的定义(比如,列的宽度,这和缓冲区应该保留多少长度有直接关系)。那么,当执行的是用户自由输入的 SQL 语句时,程序既不知道语句中会有多少个列,也不知道列的定义,该如何用 SQLBindCol 函数绑定列缓冲区呢?

我们已经知道可以用 SQLNumResultCols 函数去检测结果集中列的数量,而每个列的属性可以用 SQLDescribeCol 函数来获取,包括列的数据类型和长度等各种属性,根据这些属性,就可以决定绑定时的缓冲区类型和长度了,SQLDescribeCol 函数的用法是:

invoke	SQLDescribeCol, hStmt, dwColNumber, lpColName, dwBufferLength, \
	lpwNameLength, lpwDataType, lpdwColSize, lpwDecDigits, lpwNullable

参数中的 hStmt 是语句句柄,每次调用 SQLDescribeCol 函数可以获取一个列的属性, dwColNumber 指定列的编号,编号从 1 开始计算。

lpColName 指向一个字符串缓冲区, dwBufferLength 参数指定了这个缓冲区的长度,函数会在缓冲区中返回列的名称。lpwNameLength 则是指向一个 word 的指针,函数在这个 word 中返回实际拷贝到 lpColName 缓冲区中的字符串的长度(不包括结尾的 0 字符)。

后面的一系列参数都是指向一个 dword 或者 word 的指针,函数会在这些 dword 或者 word 中返回和列相关的属性:

- lpwDataType——在指针指向的 word 中返回列的数据类型,取值是表 18.1 中的 SQL 数据类型定义,如 SQL_CHAR、SQL_VARCHAR 等。

- `lpdwColSize`——在指针指向的 `dword` 中返回列的宽度。
- `lpwDecDigits`——如果列的数据类型是带小数位的数值，那么在这个指针指向的 `word` 中返回小数的位数。
- `lpwNullable`——在指针指向的 `word` 中返回列中是否允许空值的标志，取值可能是 `SQL_NO_NULLS` 或者 `SQL_NULLABLE`，分别表示不允许或允许空值。当驱动程序无法检测的时候，也可能返回 `SQL_NULLABLE_UNKNOWN`。

在使用 `SQLDescribeCol` 函数时，要特别注意的是参数中的指针指向的变量类型，大部分指针是指向 `word` 变量的，但也有指向 `dword` 变量的。

在实际的应用中，经常会用 `lpdwColSize` 返回的列宽度来动态申请一块内存，并将申请到的内存绑定到列，这时要特别注意的是，返回的列宽度不一定是以字节为单位的，也有可能是以 `unicode` 字符为单位。

例如，`Oracle` 数据库驱动程序返回的列宽度是以字节为单位的，但 `Access` 数据库驱动程序返回的列宽度却是 `Unicode` 字符数，如果在 `Access` 中定义了一个长度为 10 的字符型字段，用 `SQLDescribeCol` 函数检测会得到 10，但这个字段中却允许放入 10 个中文，如果我们申请 10 字节的缓冲区并绑定到列，那么返回的数据可能会被截掉尾巴，因为 10 字节的缓冲区中只能放 5 个中文字符。

用 `SQLDescribeCol` 函数检测列宽度并动态申请缓冲区的例子请参考 18.4 节中的例子。



在实际的使用中，利用 `SQLDescribeCol` 函数检测列宽度并动态申请缓冲区内存时，一般申请 `2*dwColSize+1` 长度的缓冲区，将列宽度乘以 2 是为了应对宽度的单位是 `unicode` 字符的情况，多出来的 1 字节是为了放置字符串末尾的 0 字符。如果能确定访问的数据库的列宽度是以字节为单位的，也可以将 2*去掉，这样可以节省内存。

3. 提取数据

列缓冲区绑定完毕后，我们可以根据游标的类型分别用 `SQLFetch` 或 `SQLFetchScroll` 函数来将一行的数据返回到绑定的缓冲区中。

当游标的类型是 `SQL_CURSOR_FORWARD_ONLY` 类型时，游标只能前移（增加行号）而不能后移（减少行号），这时可以用 `SQLFetch` 函数来获取数据。`SQLFetch` 函数只有一个函数——语从句柄。它首先将游标前移一行，然后将已绑定的列的数据返回到缓冲区中：

invoke `SQLFetch, hStmt`

如果函数执行成功，并且已经将数据返回到缓冲区中，那么函数返回 `SQL_SUCCESS` 或者 `SQL_SUCCESS_WITH_INFO`；如果执行成功，但是游标已经到了结果集的末尾而导致没有数据返回，那么函数返回 `SQL_NO_DATA`；当函数执行失败的时候，将根据情况返回 `SQL_ERROR` 等其他代码。

由于游标刚打开的时候位于第一行之前，所以首次执行 `SQLFetch` 函数时，游标前移一行

后刚好位于第一行，函数将返回第一行的数据，以后多次重复调用 SQLFetch 函数，即可一行一行地将结果集中的所有行返回，直到函数返回 SQL_NO_DATA 为止。

当游标的类型是 SQL_CURSOR_STATIC 类型时，游标可以自由滚动，用 SQLFetch 函数就体现不出优势来，这时我们可以用功能有所增强的 SQLFetchScroll 函数：

invoke SQLFetchScroll, hStmt, dwFetchOrientation, dwOffset

SQLFetchScroll 函数的返回值和 SQLFetch 函数相同，但比后者增加了两个参数，函数首先根据这两个参数移动游标，然后再将游标到达的行的数据返回到绑定的列缓冲区中。其中 dwFetchOrientation 参数表示移动游标的方法，参数的取值可以是以下值之一（假设函数执行前的行号为 n ，新的行号为 m ，最大行号为 \max ）：

- SQL_FETCH_NEXT——游标前移一行（ $m=n+1$ ），这时 dwOffset 参数的值被忽略。选择 SQL_FETCH_NEXT 值的时候，函数的功能和 SQLFetch 相同。
- SQL_FETCH_PRIOR——游标后移一行（ $m=n-1$ ），dwOffset 参数的值被忽略。
- SQL_FETCH_FIRST——游标移动到第一行（ $m=1$ ），dwOffset 参数的值被忽略。
- SQL_FETCH_LAST——游标移动到最后一行（ $m=\max$ ），dwOffset 参数的值被忽略。
- SQL_FETCH_ABSOLUTE——游标移动到 dwOffset 参数指定的行（ $m=dwOffset$ ）。
- SQL_FETCH_RELATIVE——游标从当前行开始，前移由 dwOffset 参数指定的行数（ $m=n+dwOffset$ ），如果 dwOffset 参数为负数，则后移 dwOffset 个记录。

在计算游标位置的过程中，如果新的行号小于 1，则游标移动到第一行之前；如果新的行号大于 \max ，则游标移动到最后一行之后，这两种情况下函数均返回 SQL_NO_DATA。

需要注意的是，当 dwFetchOrientation 的取值不是 SQL_FETCH_NEXT 的情况下，游标类型必须是可移动的，否则函数会返回错误。

4. 检测结果集中数据的行数

在实际应用中，读取结果集前经常需要先得到结果集中的记录行数，以便用来计算显示的页数等信息，在网上能够查到的很多资料中，提供的都是下面两种方法：

第一种方法是用 SQLFetch 函数遍历一遍结果集并进行计数，然后重新打开结果集（也就是重新执行 SQL 语句），这种方法有两个缺点，首先是效率问题，当结果集很大的时候，遍历结果集需要花很长时间；另外，两次执行 SQL 语句的中间，其他程序可能已经改动了记录，造成计数不准确。（当然，用动态游标可以不必重新执行语句，这时不存在第二个问题）。

第二种方法是首先用同样的 where 条件执行 “select count(*) from table where ...” 语句，得到计数值后再执行原来的查询语句，这种方法的缺点是，首先，在一个很大的表中执行计数操作可能非常慢；其次，如果执行的语句是用户自由输入的 SQL 语句，那么动态构建上述用于计数的 SQL 语句是非常困难的；最后，这种方法还是没法解决两次执行的过程中，记录可能被其他程序修改造成计数值不准确的问题。

实际上，可以用 SQLFetchScroll 函数配合 SQLGetStmtAttr 函数来检测结果集中包含的记

录行数，这是因为语句句柄的 `SQL_ATTR_ROW_NUMBER` 属性中保存有当前行号，用 `SQLFetchScroll` 函数将游标移动到最后一行后，再获取的当前行号就是结果集中的总行数：

	. data?		
dwTemp	dd	?	
dwRecRows	dd	?	;记录的总行数
	. code		
	invoke	SQLFetchScroll, hStmt, SQL_FETCH_LAST, 0	
	invoke	SQLGetStmtAttr, hStmt, SQL_ATTR_ROW_NUMBER, \	
		addr dwRecRows, SQL_IS_INTEGER, addr dwTemp	

但是并不是所有数据库的 ODBC 驱动程序都支持 `SQL_ATTR_ROW_NUMBER` 属性，例如 Access 数据库的驱动程序就不支持该属性，也就无法用这种方法检测出正确的结果集行数，但 Oracle、SQL Server 等大部分数据库驱动程序则支持 `SQL_ATTR_ROW_NUMBER` 属性，所以这种方法还是非常实用的。

读者可以根据数据库的具体情况来决定使用哪种方法检测结果集中的记录行数。

5. 结果集的释放和语句句柄的释放

在所有数据获取完毕后需要将结果集释放，这可以用 `SQLCloseCursor` 函数来实现，该函数关闭游标并释放结果集，这样语句句柄就能用来重新执行其他语句。

`SQLCloseCursor` 函数的用法是：

invoke	SQLCloseCursor, hStmt
--------	-----------------------

如果执行的语句不是 `select` 语句，那么执行结果不会产生结果集，这时在重用语句句柄之前就没有必要调用 `SQLCloseCursor` 函数。

如果语句句柄不需要被用来执行新的 SQL 语句，那么可以使用 `SQLFreeHandle` 函数来将其释放：

invoke	SQLFreeHandle, SQL_HANDLE_STMT, hStmt
--------	---------------------------------------

假如在释放语句句柄之前用 `SQLDisconnect` 函数断开了到数据库的连接，那么该连接上的所有语句句柄将被自动释放。

18.3.4 事务处理

1. 什么是事务

事务 (Transaction) 是关系型数据库中的一个重要特征，它指的是被当做单个逻辑单元来执行的一系列操作。举例来说，假如张三的账户中有 5 000 元，现在张三到银行给李四转账 1 000 元，那么转账过程至少包括下面几步和数据库相关的操作：

- 张三的账户余额减少了 1 000 元，新的余额是 4 000 元。
- 张三的账户明细记录增加一笔转出的操作。
- 李四的账户余额增加了 1 000 元。
- 李四的账号明细记录增加一笔转入的操作。

这些操作必须作为单个逻辑单元来执行,也就是说,如果转账操作成功,那么这四步数据库操作必须全部成功;如果其中的某条语句执行失败,那么数据库中被其余语句修改的数据必须全部恢复到执行前的样子,这四步操作的整体就是一个事务。

如果数据库不提供对事务的支持,那么假如在第二步完成后由于系统断电或者别的原因造成后续操作失败,就会出现张三的账户余额已被减少,而李四的账户余额不会增加的情况。时间一长,数据库中的数据必定存在很多不可预测的错误。

数据库对事务的支持表现在下面一些方面:

首先,事务将一组相关操作组合成一个要么全部成功要么全部失败的单元。程序可以在全部操作完成后选择提交(Commit)或者回滚(Rollback)事务,提交操作将事务对数据库所做的修改存盘,回滚则将整个事务所做的修改全部撤销。

其次,并发的事务之间是隔离的,假如应用程序和数据库建立了两个连接,连接 A 中执行上面的转账事务,连接 B 中进行查看,那么不管操作已经完成几步,连接 B 中看到的要么是事务开始前的数据,要么是事务结束后的数据,不能看到中间状态的数据。例如第一步执行后,在连接 A 中用 select 语句看张三的余额会是 4 000,而连接 B 中看到的还会是 5 000,只有连接 A 提交了事务,那么连接 B 中才会看到所有被修改的数据。

再次,事务间有排他的写入锁定机制,如果在事务中修改了某个表,那么对应的记录将被锁定,在该事务结束前,其他事务将无法修改被锁定的记录。锁定的范围取决于数据库的具体实现,有可能是锁定部分记录,也可能锁定整个表。

最后,一旦被提交或回滚,那就意味着事务已经结束。已经结束的事务无法再次提交或者被回滚。

现在流行的大型数据库全部支持事务机制,MySQL、dBase 等大部分小型数据库则不提供对事务的支持。在小型数据库中,Access 数据库可以支持事务,但是性能比较差,最显著的差别就在于对锁定机制的处理上,Oracle 等大型数据库采用的是行级锁,当某个事务对表中的记录进行了修改后,数据库仅锁定被修改的行,其他事务还是可以修改表中其他的行,但 Access 数据库仅实现了表级锁,也就是说即使某个事务只修改表中的一条记录,也会引起整个表被锁定,其他事务将无法修改表中的其他记录,这样在需要实现并发事务时,Access 数据库基本上不具备使用性。

2. 事务的实现

在 ODBC 接口中,可以通过设置连接句柄的 SQL_ATTR_AUTOCOMMIT 属性来决定是否打开事务机制,在默认情况下,该属性设置为 SQL_AUTOCOMMIT_ON,也就是说,每执行一条语句后,ODBC 接口会自动将语句提交,这样就相当于每条语句自动成为一个事务,程序中就无法将多条语句组合成一个事务进行统一提交或回滚了。

当属性设置为 SQL_AUTOCOMMIT_OFF 时,ODBC 接口不会将语句自动提交,需要程序在合适的时候执行 SQLEndTran 函数将前面执行的语句统一提交或者回滚。两次执行 SQLEndTran 函数之间执行的 SQL 语句就组成了一个事务。

SQLEndTran 函数的用法是:

invoke SQLEndTran, dwHandleType, hHandle, dwCompletionType

参数 dwHandleType 指定了需要操作的句柄类型, 当该参数指定为 SQL_HANDLE_ENV 时, 参数 hHandle 必须指定为一个环境句柄; 当 dwHandleType 指定为 SQL_HANDLE_DBC 时, 参数 hHandle 必须指定为一个连接句柄。

当指定的句柄是环境句柄时, 函数对通过该环境句柄分配的所有连接句柄逐一进行操作; 指定的句柄是连接句柄时, 函数仅对该连接进行提交或回滚操作。

参数 dwCompletionType 则指定了操作的类型, 如果指定为 SQL_COMMIT, 则函数将事务提交, 如果指定为 SQL_ROLLBACK, 则函数将事务回滚。

在使用 SQLEndTran 函数的时候需要注意以下事项:

首先, 如果希望在程序中实现事务, 就必须确认连接的数据库是否提供对事务的支持, 如果连接的是 MySQL 或者 dBase 等不支持事务的数据库, 那么每条语句执行后肯定会被自动提交, SQLEndTran 函数形同虚设。

其次, 事务处理的具体特征取决于数据库的具体实现, 假设我们现在运行了几条 update 语句后再运行一条 create table... 语句, 然后用 SQLEndTran 函数进行回滚操作。在 Access 数据库中执行时, SQLEndTran 函数会将前面的 update 语句连同后面的建表语句一起回滚掉。但 Oracle 数据库规定 DDL 或者 DCL 语句会在执行前后隐含执行提交操作, 所以在 Oracle 数据库中进行上述操作时, 建表语句已经将前面的 update 语句提交, 相当于结束了事务, 即使后面用 SQLEndTran 函数来回滚, 也不可能将前面已经结束的事务再回滚回去。

再次, 有些数据库中已经存在提交或回滚事务的 SQL 语句, 如 Oracle 或 SQL Server 数据库中可以通过 SQLExecDirect 函数执行 “commit” 语句来提交事务, 也可以执行 “rollback” 语句来回滚事务, 这些语句的作用和 SQLEndTran 函数没什么不同, 但 Access 等数据库中却没有这两个语句, 所以从程序的兼容性考虑, 应该总是使用 SQLEndTran 函数来结束事务。

最后, 由于 ODBC 驱动程序允许多线程操作, 所以程序可以使用同一个连接句柄来分配多个语句句柄, 然后在不同的线程中同时执行不同的 SQL 语句, 但是 SQLEndTran 函数的最小单位却是一个连接, 所以在一个线程中执行 SQLEndTran 函数, 会将使用同一个连接句柄的所有线程中的语句提交掉。如果希望在多个线程中实现互不干扰的多个事务, 这些线程必须使用独立的连接来执行 SQL 语句。

18.4 数据库操作的例子

ODBC 工作流程的各部分之间是连续的, 如环境初始化后才能连接到数据库, 连接成功后才能执行 SQL 语句, select 语句执行成功后才能读取结果集, 用到的函数相互关联, 很难割裂成多个单独的例子来演示, 所以本章将一个完整的例子放在这里统一进行演示和分析。

例子程序的代码位于 Chapter18\OdbcSample 目录中, 运行后的界面如图 18.9 所示。这是

一个能使用 ODBC 驱动程序自由连接到各种类型的数据库，并自由执行用户输入的 SQL 语句的小程序。

在“ODBC 连接字符串”文本框中输入合适的连接字符串，单击“连接”按钮后，如果输入的字符串正确并且数据库正常工作，那么程序即可连接到数据库。并激活 SQL 语句输入文本框和“执行”、“提交”、“回滚”等按钮，以便开始执行 SQL 语句。

为了方便演示，目录中已经有一个 Access 数据库 Test.mdb，并在连接字符串文本框中默认填入了“Driver={Microsoft Access Driver (*.mdb)};dbq=test.mdb”，这样读者可以马上连接到这个测试数据库进行操作。如果读者希望连接到其他数据库进行操作，也可以输入其他的连接字符串。

测试数据库 Test.mdb 中建有二个表：addr_group 表中存放通信录分组信息，表中有 id, group_name, sort 三个字段，分别记录分组 id、分组名称和排列顺序；addr 表中存放通信录，存放有 id, group_id, name, mobile, gender, company, addr, phone, post 和 memo 等字段，读者从字段的名称即可看出其含义。连接到 Test.mdb 数据库后，读者执行下面两条 SQL 语句，即可看出二个表中的数据：“select * from addr_group”，“select * from addr”。

例子程序能够连接到任何数据库，并能执行任何的 SQL 语句，当语句执行后，能自动根据语句类型显示不同的结果：如果执行的是 select 语句，程序能够自动获取结果集中各个列的名称和宽度，并根据这些信息初始化 ListView，然后在 ListView 中显示结果集中的全部数据；如果用 insert, update 等语句对记录进行修改，则能用提交或回滚按钮来将操作存盘或者撤销。总之，这个只有几百行代码的例子麻雀虽小，五脏俱全，能够实现绝大多数常用的数据库操作。

目录中的 Odbc.asm 和 Odbc.rc 文件是例子的汇编源代码和资源文件。在 Odbc.asm 中用了二个通用的模块：_ListView.asm 和 _RecordSet.asm，这两个文件中包含了一些通用的子程序，可以原封不动地包含在其他汇编源代码中使用。

_ListView.asm 模块中包含了 3 个和列表框控件相关的子程序：_ListViewAddColumn 用于在列表框中插入一个列；_ListViewSetItem 用于添加一个新的行，并对行中的数据进行修改；_ListViewClear 用于清除整个列表框中的数据。这个文件的内容在本节中就不再详细列出并分析了，有兴趣的读者请查看光盘中的文件内容并参考其中的注释进行分析。

_RecordSet.asm 模块中则包含了和结果集处理相关的子程序，这部分的功能写成通用的子程序有利于在其他程序中重复使用。

下面，我们来具体看看源代码，并对代码进行分析。

18.4.1 结果集处理模块

在使用 ODBC API 获取结果集的时候，读者一定发现了一个问题，那就是整个操作流程很烦琐——每次执行 SQL 语句后，需要根据各字段的长度分配多个缓冲区，然后多次调用 SQLBindCol 函数来将缓冲区绑定到列中，以后才能使用 SQLFetchScroll 函数获取数据。

一些应用程序需要执行的 SQL 语句很多，这样就需要定义非常多的缓冲区变量并伴随着大量

哭

```
Do Until rs.EOF
    Response.Write "姓名: " & rs(0) & " 地址: " & rs(1)
    ...
    rs.MoveNext
Loop
rs.Close
```

```
;))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

```

hStmt      dd      ?      ;执行语句用的 StateMent 句柄
dwCols     dd      ?      ;当前结果集中的列数
lpField    dd      100 dup (?) ;预留 100 个列缓冲区的指针
dwTemp     dd      ?

ODBC_RS     ends
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        assume esi:ptr ODBC_RS
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
; 释放“结果集”——释放为各字段申请的缓冲区内存
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
_RsClose    proc     uses esi ebx _lpRs

        mov     esi, _lpRs
        xor     ebx, ebx
        .while  ebx <  [esi].dwCols
            lea     eax, [esi+ebx*4+ODBC_RS.lpField]
            mov     eax, [eax]

```

720

器器

首先，代码中定义了一个 ODBC_RS 结构来存放一些中间数据，这个结构可以看成是一个对象。结构中定义了语句句柄 hStmt，结果集中列的数量 dwCols，还预留了 100 缓冲区指针，这些指针用来指向分配的缓冲区内存，其中每个列使用一个指针。结构中的 dwTemp 字段在调用 SQLFetchScroll 时临时使用。

1. _RsOpen 子程序——初始化结果集

子程序中首先用 `SQLNumResultCols` 函数获取结果集中列的数量，顺便也可以检测执行的是不是 `select` 语句，如果得到的列数为 0，则表示执行的不是 `select` 语句，不存在结果集，这时子程序直接返回错误代码。

在调用 `SQLBindCol` 函数时，为了简单起见，缓冲区的类型统一被指定成 `SQL_C_CHAR` 类型，这样 ODBC 接口返回的数据将全部是字符串类型的，可以直接用来显示。如果读者需要在缓冲区中存放实际类型的数据，那么请尝试自己修改源代码，预留一些双字变量将 `SQLDescribeCol` 函数返回的字段类型保存起来。

_RsOpen 子程序调用完毕后，主程序需要移动结果集指针时可以调用_RsMoveNext 子程序，这个子程序首先将所有预先申请的缓冲区中的数据清零，然后调用 SQLFetchScroll 函数移动指针并获取数据。

722

哭

哭

哭

哭

哭

哭

哭

哭

哭

哭

哭

资源文件中定义了如图 18.9 所示的对话框，这个对话框将被作为程序的主界面来使用，其中的 IDC_LIST 控件是一个“SysListView32”控件，这是一个列表框控件，其他的控件都是文本框和按钮等最常用的控件。

```
.386  
.model flat, stdcall  
option casemap :none ; case sensitive  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
; Include 数据  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
include windows.inc  
include user32.inc  
includelib user32.lib  
include kernel32.inc  
includelib kernel32.lib  
include comctl32.inc  
includelib comctl32.lib  
include odbc32.inc  
includelib odbc32.lib  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
; equ 数据  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
ICO_MAIN e qu 1000  
DLG_MAIN e qu 2000  
IDC_CONN_STR equ 2001  
IDC_CONN equ 2002  
IDC_DISCONN equ 2003  
IDC_SQL equ 2004  
IDC_EXEC equ 2005  
IDC_LIST equ 2006  
IDC_INFO equ 2007  
IDC_COMMIT equ 2008  
IDC_ROLLBACK equ 2009  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
; 数据段  
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>  
 .data ?  
  
hInstance dd ?  
hWinMain dd ? ;对话框句柄  
hListView dd ? ;列表框句柄
```

725

```

        invoke    SQLNumResultCols, @hStmt, addr @dwRecordCols
        and      @dwRecordCols, 0ffffh
        .if      ! @dwRecordCols
            invoke    SQLRowCount, @hStmt, addr @dwResultRows
            .if      @dwResultRows == -1      ;DDL 或 DCL 语句
                invoke    SetDlgItemText, hWinMain, \
                    IDC_INFO, addr szErrDDL
            .else
                ;DML 语句
                invoke    wsprintf, addr @szMsg, \
                    addr szErrDML, @dwResultRows
                invoke    SetDlgItemText, hWinMain, IDC_INFO, addr @szMsg
            .endif
            jmp      _FreeStmt
        .endif

;*****
; 如果是 Select 语句, 则根据结果集初始化 ListView 的标题, 以便显示
;*****
        invoke    SetDlgItemText, hWinMain, IDC_INFO, addr szErrDQL
        invoke    ShowWindow, hListView, SW_SHOW
        xor      ebx, ebx
        .while    ebx < @dwRecordCols
            inc     ebx
            invoke    SQLDescribeCol, @hStmt, ebx, \
                addr @szName, sizeof @szName, addr @dwNameSize, \
                addr @dwType, addr @dwSize, addr @dwSize1, addr @dwNullable
            mov      eax, @dwSize      ;列宽度=字符数*8 像素
            mov      ecx, 8
            mul      ecx
            .if      eax > 300      ;最大不超过 300 像素
                mov    eax, 300
            .endif
            .if      eax < 40      ;最小不小于 40 像素
                mov    eax, 40
            .endif
            lea      ecx, @szName      ;将列名称插入列表框
            invoke    _ListViewAddColumn, hListView, ebx, eax, ecx
        .endw

;*****
; 将结果集填写到 ListView 中
;*****
        invoke    _RsOpen, addr @stRs, @hStmt
        xor      esi, esi
        .while    TRUE
            invoke    _RsMoveNext, addr @stRs
            .break    .if eax
            invoke    _ListViewSetItem, hListView, esi, -1, 0      ;插入新的一行
            mov      esi, eax
            xor      ebx, ebx      ;循环显示一行中的所有列
            .while    ebx < @dwRecordCols
                invoke    _RsGetField, addr @stRs, ebx
                .if      eax
                    invoke    _ListViewSetItem, \
                        hListView, esi, ebx, eax
                .endif
            .endw
        .endw

```

器器

728


```

_ProcDlgMain proc uses ebx edi esi hWnd, wParam, lParam
    local @stWsa:WSADATA

    mov     eax, wParam
    .if     eax == WM_COMMAND
        mov     eax, wParam
;*****
; 输入连接字符串后才激活“连接”按钮
;*****
        .if     ax == IDC_CONN_STR
            invoke GetDlgItemText, hWnd, IDC_CONN_STR, \
                addr szConnString, sizeof szConnString
            invoke GetDlgItem, hWnd, IDC_CONN
            .if     szConnString
                invoke EnableWindow, eax, TRUE
            .else
                invoke EnableWindow, eax, FALSE
            .endif
;*****
; 输入 SQL 语句后才激活“执行”按钮
;*****
        .elseif ax == IDC_SQL
            invoke GetDlgItemText, hWnd, IDC_SQL, \
                addr szSQL, sizeof szSQL
            invoke GetDlgItem, hWnd, IDC_EXEC
            .if     szSQL
                invoke EnableWindow, eax, TRUE
            .else
                invoke EnableWindow, eax, FALSE
            .endif
;*****
; 连接、断开连接、执行按钮的处理
;*****
        .elseif ax == IDC_CONN
            invoke _Connect
        .elseif ax == IDC_DISCONN
            invoke _Disconnect
        .elseif ax == IDC_EXEC
            invoke _Execute
            invoke SendDlgItemMessage, hWnd, IDC_SQL, EM_SETSEL, 0, -1
        .elseif ax == IDC_COMMIT
            invoke SQLEndTran, SQL_HANDLE_DBC, hConn, SQL_COMMIT
        .elseif ax == IDC_ROLLBACK
            invoke SQLEndTran, SQL_HANDLE_DBC, hConn, SQL_ROLLBACK
        .endif
;*****
        .elseif eax == WM_INITDIALOG
            push hWnd
            pop     hWinMain
            invoke LoadIcon, hInstance, ICO_MAIN
            invoke SendMessage, hWnd, WM_SETICON, ICON_BIG, eax

            invoke GetDlgItem, hWnd, IDC_LIST
            mov     hListView, eax

```


由于调用 `SQLDriverConnect` 函数时, 使用了 `SQL_DRIVER_COMPLETE` 参数, 所以读者可以用类似于 “Driver=SQL Server” 这样的最简化的连接串来连接, 等 ODBC 驱动程序弹出参数输入框后再输入具体的参数, 然后再查看函数返回的完整的连接字符串有什么不同。

2. 执行 SQL 语句

现在可以输入 SQL 语句并执行了, 单击“执行”按钮后, 程序调用 `_Execute` 子程序。`_Execute` 子程序实现的是图 18.4 中的步骤 2、步骤 3 和步骤 4 的功能, 其中首先申请一个语句句柄, 将句柄的光标属性设置成可滚动光标后再用 `SQLExecDirect` 函数执行 SQL 语句, 由于 SQL 语句是用户自由输入的, 所以在这个例子中无法演示 `SQLPrepare`, `SQLBindParameter` 和 `SQLExecute` 等函数的用法了。

如果程序检测到语句执行失败 (返回值不是 `SQL_NO_DATA`, `SQL_SUCCESS` 或者 `SQL_SUCCESS_WITH_INFO`), 那么使用 `SQLGetDiagRec` 函数来获取具体的出错信息, 这个函数的作用类似于 Win32 API 中的 `GetLastError` 函数, 它返回的是上一条 ODBC API 的具体出错原因。

`SQLGetDiagRec` 函数的用法是这样的:

invoke	<code>SQLGetDiagRec, dwHandleType, hHandle, dwRecNumber, lpSqlstate, \</code> <code>lpdwNativeError, lpMessageText, dwBufferLength, lpdwTextLength</code>
--------	--

`dwHandleType` 参数是需要获取错误信息的句柄类型, 可以是 `SQL_HANDLE_ENV`, `SQL_HANDLE_DBC` 和 `SQL_HANDLE_STMT`, `hHandle` 参数则指定句柄。

`dwRecNumber` 指定程序从哪个状态编码开始查询错误信息, 这个参数一般指定为 1。`lpSqlState` 指向一个字符串缓冲区, 函数在这里返回错误代码字符串; `lpdwNativeError` 参数指向一个双字, 函数在这里返回错误代码; `lpMessageText` 参数指向一个字符串缓冲区, 函数在这里返回具体出错信息字符串, `dwBufferLength` 指定了这个缓冲区的长度, 最后一个参数指向一个双字, 函数返回出错信息字符串后, 字符串的长度将被填写到这个双字中。

函数返回了三种错误信息, 这些错误信息有什么不同呢? 其实这些信息分两类, 一类是 ODBC 规范定义的标准错误信息, 另一类是驱动程序返回的本地化错误信息。

`lpSqlState` 指向的缓冲区中返回的是 ODBC 规范定义的标准错误信息, 这是一个 5 字节长的字符串, 当在不同类型的数据库中进行操作并出现同类型的错误时, 这里的代码是相同的, 比如, 01004 代表 Data truncated, 01S02 代表 Option value changed, HY008 代表 Operation canceled 等, 相同的编码有利于书写通用的错误处理代码, 详细的代码列表请读者参考 MSDN 中的相关部分。

`lpdwNativeError` 和 `lpMessageText` 指向的变量中返回的则是数据库本地化的数值型错误代码, 以及具体的出错原因字符串。比如, 同样执行 “select * from add” 语句 (假设 add 表是不存在的), 操作 Access 数据库时, 返回的本地错误代码是 -1305, 返回的出错原因字符串是 “[Microsoft] [ODBC Microsoft Access Driver] Microsoft Jet 数据库引擎找不到输入表或查询 'add'。确定它是否存在, 以及它的名称的拼写是否正确”; 而操作的是 Oracle 数据库时, 返回的本地错误代码却是 903, 返回的出错原因字符串变成了 “[oracle][ODBC][Ora]

ORA-00903: 无效表名”。

显然,本地化的出错原因字符串能提供更详细的信息,另外,在有些情况下也必须用到本地化的错误编码,例如使用 SQL Server 或 Oracle 等 C/S 数据库时,遇到网络断线后需要重连数据库,否则后面的然后操作肯定会失败,但是仅仅从标准错误信息不足以检测错误是否是由网络引起的,这时就要用到本地化的出错代码了。在实际使用中,如果网络断线,Oracle 数据库可能出现 3113 或 3114 错误,SQL Server 数据库可能出现 232 和 10054 错误,这种情况就需要根据具体的数据库进行具体处理了。

现在回过头来看 _Execute 子程序,语句执行成功的话,程序首先检测执行的 SQL 语句类型,用到的代码就是 18.3.2 节最后部分示范的代码,如果检测到是 DML 语句,程序显示语句修改的记录行数;如果是 DDL 或 DCL 等语句,则仅仅显示语句是否执行成功的信息;如果执行的是 select 语句,则获取结果集并进行处理。

要显示结果集的话,需要先用 ShowWindow 函数将隐藏的列表框控件显示出来,并根据列名在列表框中添加对应的标题,子程序中用一个循环来完成这个功能,循环中用 SQLDescribeCol 函数获得每个列的名称和宽度。然后用 _ListView.asm 文件中的 _ListViewAddColumn 子程序添加列。

现在,终于可以用到 _RecordSet.asm 文件中的那些子程序来显示整个结果集了,这部分的代码如下:

```

invoke  _RsOpen, addr @stRs, @hStmnt
xor     esi, esi                ; esi 作为行号, 从 0 开始计算
.while  TRUE
    invoke  _RsMoveNext, addr @stRs
    .break  .if eax
    invoke  _ListViewSetItem, hListView, esi, -1, 0    ; 插入新的一行
    mov     esi, eax
    xor     ebx, ebx            ; 获取并显示行中所有的列
    .while  ebx < @dwRecordCols
        invoke  _RsGetField, addr @stRs, ebx
        .if     eax
            invoke  _ListViewSetItem, hListView, esi, ebx, eax
        .endif
        inc     ebx
    .endw
    inc     esi                ; 行号加 1
.endif
invoke  _RsClose, addr @stRs

```

读者可以看到,使用了封装的子程序后,代码的结构简洁程度可以与 VBScript 程序相媲美了,原先烦杂的缓冲区分配工作和 SQLBindCol 函数调用不见了踪影。

在 _Execute 子程序的最后,程序使用 SQLCloseCursor 函数关闭结果集,并调用 SQLFreeHandle 函数释放语句句柄。

3. 事务处理

执行了一系列的 DML 语句后,可以用“提交”或“回滚”按钮来结束事务。这部分功能由

这两个按钮的处理代码实现：

```

mov     eax, wParam
.if     eax == WM_COMMAND
mov     eax, wParam
.if     ax == ...
        ; 其他按钮的处理代码
.elseif ax == IDC_COMMIT
        invoke SQLEndTran, SQL_HANDLE_DBC, hConn, SQL_COMMIT
.elseif ax == IDC_ROLLBACK
        invoke SQLEndTran, SQL_HANDLE_DBC, hConn, SQL_ROLLBACK
.endif

```

这部分的实现非常简单，只是用 SQL_COMMIT 或者 SQL_ROLLBACK 参数来调用 SQLEndTran 函数而已。

读者可以用下面的序列来测试事务功能：

(1) 两次运行 Odbc.exe 文件，这样屏幕上会有两个操作窗口。在两个窗口中执行 “select * from addr where id=1” 语句，可以发现 name 字段的值都是 “张三”。

(2) 现在在窗口 A 中执行 “update addr set name='aaa' where id=1” 后，然后在同一窗口中用步骤 1 的 select 语句查看结果，可以发现 name 字段的值变成了 “aaa”。但是在窗口 B 中再次执行步骤 1 的 select 语句，可以发现 name 字段的值还是 “张三”，说明事务结束之前，其他连接中是看不到中间结果的。

(3) 在窗口 B 中执行 “update addr set name='aaa' where id=1”，片刻后程序会提示错误：由于表被锁定造成记录无法更新，说明事务结束前，DML 语句将锁定记录。将语句中的 where 条件改为 id=2，程序还是会提示同样的错误，说明 Access 数据库实现的是表级锁。

(4) 在窗口 A 中单击 “提交” 按钮，然后在两个窗口中再次查询，name 字段的值全部变成了 “aaa”。

读者可以做个实验，将 _Connect 子程序中设置连接句柄属性的 SQLSetConnectAttr 语句去掉，重新编译链接后再进行上述操作，可以发现每条语句执行后，所做的修改都会被自动提交。

4. 断开连接

如果用户单击 “断开” 按钮，那么程序调用 _Disconnect 子程序来断开连接，子程序中首先将事务提交，再用 SQLDisconnect 函数断开到数据库的连接，然后释放连接句柄和环境句柄，最后在设置相关按钮的状态后，子程序返回。

参 考 文 献

- 1 Matt Pietrek. Windows 95 System programming SECRETS. IDG Books, 1995
- 2 Jeffrey Richter. Programming Applications for Windows. Microsoft Press, 1999
- 3 Charles Petzold. Programming Windows 95. Microsoft Press, 1996
- 4 Charles Petzold. Programming Windows. Microsoft Press, 1998
- 5 Anthony Jones & Jim Ohlund. Network Programming for Microsoft Windows. Microsoft Press, 1999
- 6 W.Richard Stevens. TCP/IP 详解: 卷 1——协议. 北京: 机械工业出版社
- 7 Microsoft ODBC 3.0 程序员参考及 SDK 指南. Microsoft Press, 1999
- 8 侯捷. 深入浅出 MFC. 华中理工大学出版社, 1998
- 9 施炜, 李铮, 秦颖. Windows Sockets 规范及应用
- 10 北京科海. 80386 系统设计手册, 1990
- 11 吕晓庆. 80386/486 系统编程实践. 浙江大学出版社, 1993
- 12 Art of Assemble. <http://www.cs.ucr.edu/~rhyde>, 1996
- 13 Iczelion's Win32asm tutorial, <http://win32asm.cjb.net>
- 14 hutch's home page, <http://www.movsd.com>

本书的编写过程中还参考了本人长期积累的、来自因特网上的编程资料, 出处一时无法考证, 在此一并表示感谢!

《琢石成器——Windows 环境下 32 位汇编

语言程序设计》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更

透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定

期的信息反馈。

1. 短信

您只需编写如下短信：B08663+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话

（010）88254369。

2. 电子邮件

您可以发邮件至jsj@phei.com.cn**或**editor@broadview.com.cn**。**

3. 信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意

见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；



www.phei.com.cn
www.broadview.com.cn



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview[®]
www.broadview.com.cn

(3) 您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B08663+退订”至10666666789即可，

或者编写邮件“B08663 +退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 亦可

取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站(www.broadview.com.cn)上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路173信箱 博文视点(100036) 电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396；(010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036